BRUNO SENS CHANG

IMPLEMENTAÇÃO EM FPGA DE TÉCNICAS DE EQUALIZAÇÃO ADAPTATIVA UTILIZANDO O ALGORITMO CORDIC

FLORIANÓPOLIS

2008

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

IMPLEMENTAÇÃO EM FPGA DE TÉCNICAS DE EQUALIZAÇÃO ADAPTATIVA UTILIZANDO O ALGORITMO CORDIC

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a obtenção do
grau de Mestre em Engenharia Elétrica.

BRUNO SENS CHANG

Florianópolis, Fevereiro de 2008

IMPLEMENTAÇÃO EM FPGA DE TÉCNICAS DE EQUALIZAÇÃO ADAPTATIVA UTILIZANDO O ALGORITMO CORDIC

Bruno Sens Chang

"Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia Elétrica, Área de Concentração em Comunicação e Processamento de Sinais, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina"

Prof. Carlos Aurélio Faria da Rocha, D. Sc.

Orientador

Prof. Kátia Campos de Almeida, Ph. D.

Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Leonardo Silva Resende, D.Sc.

Presidente

Prof. Bartolomeu Uchôa Filho, Ph. D

Prof. Raimes Moraes, Ph.D

Prof. Walter Pereira Carpes Junior, Dr.

para minha família e Thays

Agradecimentos

Muitíssimo obrigado.

À minha família, pelo carinho e apoio para a realização do mestrado.

À minha namorada Thays, pelo amor e companheirismo.

Ao meu orientador, Prof. Carlos Aurélio Faria da Rocha, pela orientação que tornou possível esta dissertação.

Aos amigos do GPqCom, especialmente a Cesar Humberto Vidal Vargas e Francisco José de Alves Aquino, pelo apoio, amizade e companheirismo.

A todos os meus amigos, por entenderem as minhas ausências.

À CAPES e à UFSC, pelo apoio financeiro e pela estrutura.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para a obtenção do grau de Mestre em Engenharia Elétrica.

IMPLEMENTAÇÃO EM FPGA DE TÉCNICAS DE EQUALIZAÇÃO ADAPTATIVA UTILIZANDO O ALGORITMO CORDIC

Bruno Sens Chang

Fevereiro/2008

Orientador: Carlos Aurélio Faria da Rocha, D. Sc.
Area de Concentração: Comunicações e Processamento de Sinais
Palavras-chaves: CORDIC, equalização adaptativa, FPGA, procedimento multisplit.
Número de páginas: 71

Esta dissertação apresenta uma implementação em uma FPGA (Arranjo de Portas Programáveis em Campo - em inglês, *Field Programmable Gate Array*) de um equalizador adaptativo que utiliza o algoritmo LMS trigonométrico. Este algoritmo utiliza variáveis trigonométricas, que são relacionadas monotonicamente aos coeficientes do equalizador. O algoritmo LMS trigonométrico é especialmente apropriado para uma realização utilizando processadores CORDIC (Computador Digital de Rotação de Coordenadas - em inglês, *Coordinate Rotation Digital Computer*), já que estes calculam simultaneamente as funções trigonométricas necessárias às operações de filtragem e adaptação dos coeficientes. A utilização deste algoritmo impõe a necessidade da definição de um hipercubo, no qual deve estar contido o ponto de mínimo da função objetivo (potência do erro entre o sinal desejado e a saída do equalizador). A literatura disponível não é clara quanto à definição deste hipercubo. Esta dissertação propõe a adoção do procedimento multisplit para facilitar a definição desse hipercubo. Simulações realizadas indicam que a utilização do procedimento multisplit permite que o algoritmo LMS trigonométrico alcance a convergência em uma variedade de canais sem a necessidade de uma busca exaustiva dos valores das coordenadas do hipercubo.

Abstract of Dissertation presented to UFSC in partial fulfillment of the requirements for the degree of Master in Electrical Engineering

FPGA IMPLEMENTATION OF ADAPTIVE EQUALIZATION TECHNIQUES USING THE CORDIC ALGORITHM

Bruno Sens Chang

February/2008

Advisor: Carlos Aurélio Faria da Rocha, D. Sc.
Area of Concentration: Communications and Signal Processing
Keywords: CORDIC, adaptive equalization, FPGA, multisplit procedure.
Number of pages: 71

This dissertation presents a FPGA (Field Programmable Gate Array) implementation of an adaptive equalizer using the trigonometric LMS algorithm. This algorithm uses trigonometric variables, which are monotonically related to the equalizer coefficients. The trigonometric LMS algorithm is specially appropriate for a realization using CORDIC (Coordinate Rotation Digital Computer) processors, since they compute simultaneously the trigonometric functions needed for filtering and coefficient updating. When using this algorithm, the need to define a hypercube which contains the minima of the objective function (error power between the desired signal and equalizer output) arises. Available literature is not clear regarding the hypercube definition process. This dissertation proposes the adoption of the multisplit procedure to facilitate the hypercube definition. Simulation results show that the multisplit procedure allows the trigonometric LMS algorithm to converge in a variety of channels without the need for an exaustive search for the hypercube coordinate values.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ALF *Adaptive Lattice Filter*

AM *Amplitude Modulation*

AR *Autoregressive*

ARMA *Autoregressive Moving Average*

ASK *Amplitude Shift Keying*

ASR *Address Shift Register*

CALF *CORDIC Adaptive Lattice Filter*

CIC *Cascaded Integrator Comb*

CORDIC *Coordinate Rotation Digital Computer*

DAC *Digital-to-Analog Converter*

DDS *Direct Digital Synthesis*

DFT *Discrete Fourier Transform*

DLMS *Delayed Least Mean Squares Algorithm*

DSP *Digital Signal Processor*

FFT *Fast Fourier Transform*

FHT *Fast Hadamard Transform*

FM *Frequency Modulation*

FPGA *Field Programmable Gate Array*

FSK *Frequency Shift Keying*

GLF *Gradient Lattice Filter*

GSC *Generalized Sidelobe Canceller*

HDL *Hardware Description Language*

IES Interferência entre Símbolos

LMS *Least Mean Squares*

LUT *Lookup Table*

MAC *Multiply Accumulate*

ODF *Orthogonal Digital Filter*

PLL *Phase Locked Loop*

PM *Phase Modulation*

PSK *Phase Shift Keying*

QAM *Quadrature Amplitude Modulation*

RAM *Random Access Memory*

RLSLF *Recursive Least Squares Lattice Filter*

ROM *Read Only Memory*

SFDR *Spurious-Free Dynamic Range*

WHT *Walsh-Hadamard Transform*

Capítulo 1

Introdução

Atualmente, a comunicação usando sinais elétricos está tão integrada às nossas vidas que torna-se muito fácil ignorar a grande variedade de aplicações que ela possibilita. É possível definir um sistema de comunicação como um sistema que tem como meta transmitir a informação de um transmissor a um receptor.

O principal problema em um sistema de comunicação é o surgimento das interferências entre símbolos (IES) devido à transmissão do sinal por um canal de comunicação não ideal. Tais interferências corrompem a mensagem transmitida, sendo necessária a utilização de um dispositivo de compensação no receptor. Filtros equalizadores são comumente utilizados para compensar a IES.

Desde seu surgimento, em 1958, o algoritmo CORDIC (*Coordinate Rotation Digital Computer*) é utilizado em várias aplicações de comunicações e processamento de sinais, devido a sua eficiência na solução de problemas trigonométricos [1]. Entretanto, para filtros adaptativos, a utilização do algoritmo CORDIC era limitada aos filtros em treliça, já que os cálculos, a cada estágio desse tipo de filtro, podem ser diretamente relacionados a rotações angulares [2, 3].

Para filtros adaptativos transversais, o algoritmo LMS trigonométrico foi introduzido por Chakraborty *et al.* em 2005 [4]. Nesse algoritmo, os cálculos necessários para as operações de filtragem e atualização dos coeficientes podem ser realizados simultaneamente por processadores CORDIC. No entanto, o artigo citado não é claro quanto ao método de definição das coordenadas do hipercubo que contém o ponto de

mínimo da função objetivo.

1.1 Objetivo

O objetivo desta dissertação é a implementação em uma FPGA (Field Programmable Gate Array) de um equalizador adaptativo que utiliza o algoritmo LMS trigonométrico com o procedimento multisplit. Este procedimento aumenta a potência do sinal transformado, diminuindo assim os valores dos coeficientes do equalizador e tornando possível a convergência do algoritmo LMS trigonométrico com um hipercubo fixo.

As FPGAs têm sido freqüentemente utilizadas em sistemas de comunicações devido à sua grande flexibilidade, eficiência e possibilidade de reprogramação dinâmica. Na implementação do equalizador utilizou-se uma FPGA Virtex-4 produzida pela Xilinx.

1.2 Organização do Trabalho

A seguir, discute-se com mais detalhes o que será apresentado nessa dissertação. O Capítulo 2 contém uma breve introdução sobre o algoritmo CORDIC. Esta revisão inclui o histórico, a dedução das três equações fundamentais do algoritmo e os métodos para aumentar a faixa de operação do algoritmo.

O Capítulo 3 apresenta uma revisão bibliográfica sobre as aplicações do algoritmo CORDIC em comunicações e processamento de sinais. Entre as aplicações revistas, podem-se citar a síntese digital direta (em inglês, Digital Direct Synthesis - DDS), sincronização, modulação, *up/downconversion*, a transformada rápida de Fourier (em inglês, Fast Fourier Transform - FFT) e filtragem digital.

Já o Capítulo 4 aborda a equalização adaptativa trigonométrica multisplit. O algoritmo LMS trigonométrico, o procedimento multisplit e a transformada de Hadamard utilizada pelo procedimento multisplit são detalhados. Ao final do capítulo, apresenta-se uma análise das vantagens que o procedimento multisplit confere ao algoritmo LMS

trigonométrico.

O Capítulo 5 expõe detalhes sobre a ferramenta utilizada para a implementação dos equalizadores em FPGA: o System Generator, da Xilinx. Detalhes sobre o fluxo de projeto, os blocos disponíveis para o projetista e suas propriedades de temporização e representação aritmética são fornecidos. O capítulo termina com considerações sobre a geração de código e co-simulação em *hardware* dos projetos realizados em System Generator.

O Capítulo 6 apresenta as estruturas dos processadores CORDIC e dos equalizadores implementados via System Generator. Neste capítulo são também apresentados e discutidos os resultados das simulações feitas.

Finalmente, o Capítulo 7 apresenta uma conclusão geral do estudo realizado, ressaltando as principais contribuições dessa dissertação e propondo direções para os trabalhos futuros.

Capítulo 2

CORDIC

A unidade básica da maioria dos sistemas de processamento digital de sinais (DSP - *Digital Signal Processing*) é a unidade de multiplicação e acumulação (MAC - *multiply and accumulate*). Estas operações são, geralmente, a base dos algoritmos de DSP. A otimização destes algoritmos levou a equações que necessitam do cálculo de funções que não são calculadas eficientemente através do uso de unidades aritméticas baseadas em MAC, tais como funções trigonométricas, exponenciais, logarítmicas, entre outras.

Já o algoritmo CORDIC (Computador Digital de Rotação de Coordenadas - em inglês, *Coordinate Rotation Digital Computer*) oferece uma formulação que, com apenas pequenas modificações, pode calcular de maneira eficiente cada uma destas funções. Este capítulo traz um breve histórico do desenvolvimento do algoritmo CORDIC, suas principais equações, modos de operação e maneiras de otimizar sua faixa de convergência.

2.1 Histórico

O CORDIC foi desenvolvido por Jack E. Volder, em 1956, para substituir os computadores analógicos do sistema de navegação do bombardeiro B-58 por computadores digitais, já que os analógicos eram pouco precisos. Tais computadores analógicos tinham como tarefa calcular relações trigonométricas necessárias para a navegação sobre uma Terra esférica [5]. Os dois modos principais de operação deste computador eram

o de rotação das coordenadas do vetor de entrada e o de determinação da amplitude e do ângulo de um vetor. A Figura 2.1 representa estes modos de operação dos computadores analógicos.

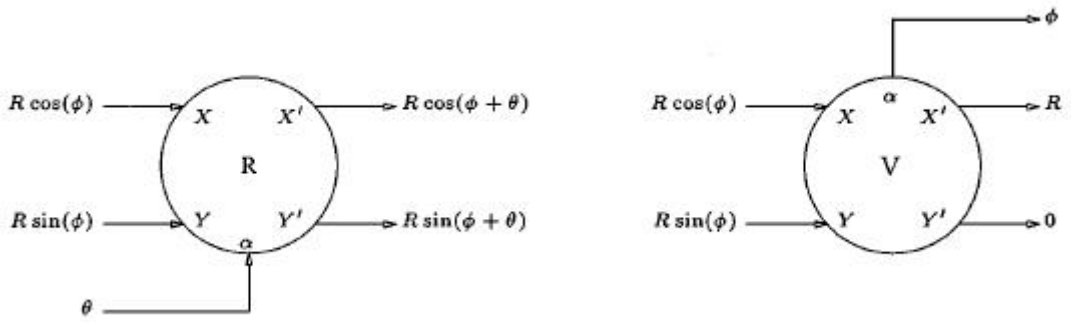

Figura 2.1: Modos de operação do computador analógico [5]

Na época, não existiam computadores digitais capazes de realizar as mesmas operações dos computadores analógicos. Volder buscou inspiração na edição de 1946 do Handbook of Chemistry and Physics para a tarefa. Modificando as equações básicas de adição de ângulos, Volder atingiu um conjunto de equações (que será exposto mais adiante) capaz de desempenhar as mesmas tarefas do cálculo analógico, mas com uma precisão muito maior.

Walther [6] foi o primeiro a estender o trabalho de Volder, a fim de resolver outros problemas, tais como relações hiperbólicas e sistemas lineares. Outros seguiram no seu caminho, e muitas propostas de utilização dos modos de operação do CORDIC foram apresentadas. Entre as aplicações mais conhecidas do CORDIC, além do protótipo inicial de Volder para navegação, é possível citar a calculadora HP-35, o coprocessador matemático 8087, entre outros.

2.2 Algoritmo

O CORDIC é um algoritmo que utiliza um método iterativo de rotações vetoriais, utilizando apenas deslocamentos e adições [7]. Tal característica o torna muito interessante pela fácil implementação em *hardware*, especialmente naqueles sem multiplicadores (como muitos FPGAs).

O algoritmo é derivado das equações gerais de rotação de um vetor com coordenadas (x,y) por um ângulo ϕ qualquer, como é representado na Figura 2.2:

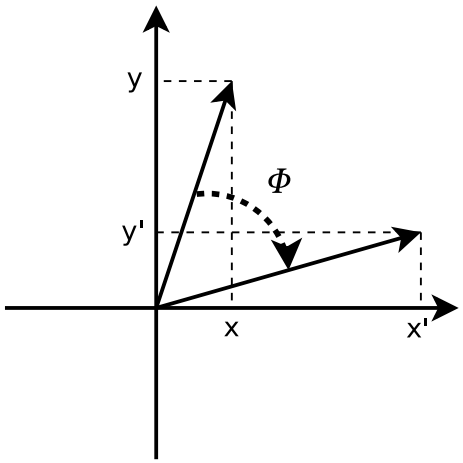

Figura 2.2: Rotação Vetorial

Esta rotação vetorial pode ser expressa por:

$$x^{'} = x\cos\phi - y\sin\phi, \tag{2.1}$$

$$y^{'} = y\cos\phi + x\sin\phi, \tag{2.2}$$

onde x' e y' são as novas coordenadas.

Tais equações podem ser modificadas para atingirem a seguinte forma:

$$x^{'} = \cos\phi\left[x - y\tan\phi\right], \tag{2.3}$$

$$y^{'} = \cos\phi\left[y + x\tan\phi\right]. \tag{2.4}$$

Definindo-se x_0 e y_0 como as coordenadas atuais, um algoritmo iterativo pode ser obtido da seguinte maneira:

$$x_1 = \cos\phi_0(\overbrace{x_0 - y_0\cdot\tan\phi_0}^{x_1^{'}}), \tag{2.5}$$

$$y_1 = \cos\phi_0(\overbrace{y_0 + x_0\cdot\tan\phi_0}^{y_1^{'}}), \tag{2.6}$$

$$x_2 = \cos\phi_1(x_1 - y_1\cdot\tan\phi_1) = cos\phi_0\cdot cos\phi_1[(\overbrace{x_0 - y_0\cdot\tan\phi_0}^{x_1^{'}}) - (\overbrace{y_0 + x_0\cdot\tan\phi_0}^{y_1^{'}})\cdot\tan\phi_1], \tag{2.7}$$

$$y_2 = \cos\phi_1(y_1 + x_1 \cdot \tan\phi_1) = cos\phi_0 \cdot cos\phi_1[(\overbrace{y_0 + x_0 \cdot \tan\phi_0}^{y_1'}) - (\overbrace{x_0 - y_0 \cdot \tan\phi_0}^{x_1'}) \cdot \tan\phi_1]. \tag{2.8}$$

Assim, pode-se reescrever as equações 2.7 e 2.8 como:

$$x_2 = \cos\phi_0 \cdot \cos\phi_1(x_1 - y_1 \tan\phi_1), \tag{2.9}$$

$$y_2 = \cos\phi_0 \cdot \cos\phi_1(y_1 + x_1 \tan\phi_1). \tag{2.10}$$

Generalizando-se, as equações 2.9 e 2.10, obtém-se:

$$x_{i+1} = \cos\phi_0 \cdot \cos\phi_1 \ldots \cos\phi_i(x_i - y_i \tan\phi_i), \tag{2.11}$$

$$y_{i+1} = \cos\phi_0 \cdot \cos\phi_1 \ldots \cos\phi_i(y_i + x_i \tan\phi_i). \tag{2.12}$$

Definindo-se $k_i = cos\phi_0 \cdot cos\phi_1 \cdot \ldots cos\phi_i$, pode-se reescrever as equações 2.11 e 2.12 como:

$$x_{i+1} = k_i \cdot (x_i - y_i \tan\phi_i), \tag{2.13}$$

$$y_{i+1} = k_i \cdot (y_i + y_i \tan\phi_i). \tag{2.14}$$

Limitando $\tan\phi_i$ a $\pm 2^{-i}$, é possível resolver estas equações utilizando apenas operações de soma e deslocamento. Esta é a relação fundamental na qual o algoritmo CORDIC é baseado. Ângulos quaisquer, dentro de uma certa precisão, podem ser formados por sucessivas rotações cada vez menores. Assim, chega-se nas seguintes equações:

$$x_{i+1} = k_i \left[x_i - y_i \cdot 2^{-i}\right], \tag{2.15}$$

$$y_{i+1} = k_i \left[y_i + x_i \cdot 2^{-i}\right], \tag{2.16}$$

$$k_i = \cos\phi_0 \cdot \cos\phi_1 \ldots \cos\phi_i, \tag{2.17}$$

$$\cos\phi_i = \frac{1}{\sqrt{1 + \tan^2\phi_i}}, \tag{2.18}$$

$$k_i = \prod_i \frac{1}{\sqrt{1 + 2^{-2i}}}. \tag{2.19}$$

Os valores de 2^{-i} podem ser pré-calculados e armazenados em uma tabela ou calculados no instante da iteração. A primeira alternativa é um pouco mais eficiente

computacionalmente, mas requer maior espaço em memória. O fator k depende do número de iterações; quando $n \to \infty$, seu valor é 0,607253. Este fator pode ser tratado como um ganho, e compensado antes ou depois das iterações.

Para que não haja rotações desnecessárias, é necessário somar os ângulos das iterações já realizadas. Este processo é feito com outro somador, que adiciona os ângulos correspondentes às rotações calculadas. Tal somador acrescenta a seguinte equação ao algoritmo CORDIC:

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i}). \tag{2.20}$$

Note que $\phi_0 = \pi/4$, e dependendo do valor do ângulo de rotação desejado, o próximo valor do ϕ_i pode seguir o sentido horário ou anti-horário. Para isso devemos alterar as equações de iterações de x'_{i+1} e y'_{i+1}, a fim de satisfazer essa restrição.

A alteração é a seguinte:

$$x_{i+1} = k_i \left[x_i - y_i \cdot d_i \cdot 2^{-i} \right], \tag{2.21}$$

$$y_{i+1} = k_i \left[y_i + x_i \cdot d_i \cdot 2^{-i} \right], \tag{2.22}$$

onde $d_i = \pm 1$. O sinal + ou - de d_i vai depender do sinal de z_i, com

$$z_{i+1} = z_i - d_i \cdot \tan^{-1} 2^{-i}, \tag{2.23}$$

$$d_i = 1 \text{ se } z_i > 0, \tag{2.24}$$

$$d_i = -1 \text{ se } z_i < 0, \tag{2.25}$$

onde z_0 é o ângulo de rotação final.

Existem dois modos básicos de operação do CORDIC: o de *rotação* e o de *vetorização*. No modo de rotação, as coordenadas de um vetor e um ângulo de rotação são fornecidas, e as coordenadas do vetor original, depois da rotação pelo ângulo dado, são calculadas. No modo de vetorização, as coordenadas do vetor são dadas e a magnitude e o ângulo deste vetor são calculados. Por conseqüência, o modo de vetorização realiza a conversão de coordenadas cartesianas para polares [1].

As equações do algoritmo CORDIC para o modo de rotação são:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}, \tag{2.26}$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}, \tag{2.27}$$

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i}), \tag{2.28}$$

onde

$$d_i = 1 \text{ se } z_i > 0, -1 \text{ se } z_i < 0. \tag{2.29}$$

Estas equações têm o seguinte resultado final:

$$x_n = A_n \left[x_0 \cdot \cos z_0 - y_0 \cdot sen z_0\right], \tag{2.30}$$

$$y_n = A_n \left[y_0 \cdot \cos z_0 + x_0 \cdot sen z_0\right], \tag{2.31}$$

$$z_n = 0, \tag{2.32}$$

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}. \tag{2.33}$$

Fazendo com que $y_0 = 0$, é possível reduzir os resultados do modo de rotação a:

$$x_n = A_n \cdot x_0 \cdot \cos z_0, \tag{2.34}$$

$$y_n = A_n \cdot y_0 \cdot sen z_0. \tag{2.35}$$

Impondo $x_0 = 1/A_n$, o processo de rotação gera o seno e o cosseno do ângulo em z_0. Uma extensão lógica destes cálculos é a conversão de coordenadas polares para retangulares. A transformada de coordenadas polares para retangulares é definida por:

$$x = r \cdot \cos(\theta), \tag{2.36}$$

$$y = r \cdot sen(\theta). \tag{2.37}$$

Estas relações são obtidas pelo algoritmo CORDIC, bastando colocar $x_0 = r$, $y_0 = 0$, e $z_0 = \theta$.

Outras técnicas de obtenção simultânea dos valores do seno e do cosseno (ex.: uma *lookup table*) necessitam de dois multiplicadores para o cálculo das operações trigonométricas. Já utilizando o modo de operação de rotação do CORDIC, é possível eliminar estes multiplicadores. Em termos de complexidade de *hardware* (células lógicas ocupadas), um rotator serial CORDIC é aproximadamente equivalente a um único multiplicador que trabalhe com o mesmo tamanho de palavra [7].

O modo de vetorização tem como objetivo rotacionar o vetor pelo ângulo que for necessário para que ele fique sobreposto ao eixo de coordenadas x. A cada iteração, este modo procura minimizar o componente y do vetor residual; este componente também determina a direção de rotação do vetor. As equações para o modo de vetorização do CORDIC são:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}, \tag{2.38}$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}, \tag{2.39}$$

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i}), \tag{2.40}$$

onde

$$d_i = 1 \text{ se } y_i > 0, -1 \text{ se } y_i < 0. \tag{2.41}$$

Estas equações têm o seguinte resultado:

$$x_n = A_n \sqrt{x_0^2 + y_0^2}, \tag{2.42}$$

$$y_n = 0, \tag{2.43}$$

$$z_n = \tan^{-1} \frac{y_0}{x_0}, \tag{2.44}$$

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}. \tag{2.45}$$

Os dois modos de operação do CORDIC funcionam somente na faixa de $-\pi/2$ a $\pi/2$, devido ao uso de 2^0 para a tangente na primeira iteração [7]. Para que o algoritmo possa ser utilizado em condições reais, é necessário estender esta faixa de operação. Por exemplo, a fim de operar como um conversor fase-amplitude em um DDS (dispositivo de síntese digital direta, que será melhor detalhado no próximo capítulo), precisa-se de uma faixa de $\pm\pi$. Existem várias propostas visando resolver este problema.

A primeira foi proposta por Volder [1], e consiste em uma rotação inicial por $\pm\pi/2$. Com esta rotação, a faixa dinâmica aumenta de π para 2π, atingindo assim todos os ângulos. Tal rotação pode ser formulada da seguinte maneira:

$$x^{'} = -d \cdot y, \tag{2.46}$$

$$y^{'} = d \cdot x, \tag{2.47}$$

$$z^{'} = z + d \cdot \pi/2, \tag{2.48}$$

$$d = +1 \text{ se } y < 0, -1 \text{ caso contrário .} \tag{2.49}$$

Deve-se notar que não há crescimento nesta primeira rotação.

Uma outra maneira utiliza as seguintes identidades trigonométricas:

$$\cos(Z - \pi) = -\cos(Z), \tag{2.50}$$

$$sen(Z - \pi) = -sen(Z). \tag{2.51}$$

Tais identidades permitem o cálculo de um ângulo fora da faixa de operação pelo seu ângulo complementar [8]. Por exemplo, quando for necessário o cálculo do valor de $\cos(3\pi/2)$, aplica-se a identidade:

$$\cos(3\pi/2 - \pi) = -\cos(3\pi/2), \tag{2.52}$$

$$\cos(\pi/2) = -\cos(3\pi/2). \tag{2.53}$$

Como é possível calcular o primeiro termo utilizando um processador CORDIC convencional, é possível descobrir o valor do segundo termo. Este método necessita de *hardware* (comparadores e corretores de ângulo) adicional para funcionar corretamente.

As duas alternativas têm o mesmo resultado final, e também uma complexidade computacional muito similar. Entretanto, a segunda apresenta uma melhor SFDR (*spurious free dynamic range* - faixa dinâmica livre de espúrios, conceito que será mais bem explorado no próximo capítulo) que a primeira, devido ao fato de que ela leva a uma melhor precisão do algoritmo na sua faixa nativa. Por exemplo, um DDS com 8 bits de precisão consegue uma SFDR de 60 dB utilizando o primeiro método e 63 dB utilizando o segundo [8].

2.3 Conclusão

Este capítulo apresentou o histórico do algoritmo CORDIC, a dedução de suas equações, seus modos de operação e os métodos que podem ser usados para aumentar a faixa de operação do algoritmo. O próximo capítulo apresentará algumas das aplicações do CORDIC em sistemas de comunicações e processamento de sinais.

Capítulo 3

Aplicações do algoritmo CORDIC

O capítulo anterior apresentou uma introdução do algoritmo CORDIC. Este capítulo apresenta algumas aplicações deste algoritmo. Informações mais detalhadas serão fornecidas sobre as seguintes aplicações: síntese digital direta (*Direct Digital Synthesis* - DDS), sincronização, modulação, *up/downconversion*, FFT (Transformada Rápida de Fourier) e filtragem digital.

3.1 Síntese Digital Direta

A síntese digital direta (DDS) é um método de geração de formas de onda diretamente no domínio digital [8]. Primeiramente proposto em 1971 por Tierney *et al* [9], o sintetizador digital está sendo usado cada vez mais nos projetos de circuitos, devido às suas várias vantagens, entre elas pode se destacar a possibilidade de obter uma resolução tão pequena quanto o projetista desejar, alta estabilidade (dispensando controle automático de ganho) e a continuidade de fase mesmo quando a freqüência muda. Além disso, tanto a fase quanto a freqüência da forma de onda podem ser controladas em um período de amostragem, possibilitando assim modulação de fase. Sua implementação pode ser realizada apenas com aritmética de números inteiros, fazendo com que seja possível a implementação de um DDS em qualquer microcontrolador, DSP ou FPGA.

Um gerador DDS é composto por um acumulador de fase e um conversor fase-amplitude, conforme mostrado na Figura 3.1. Em DDSs convencionais, o acumulador

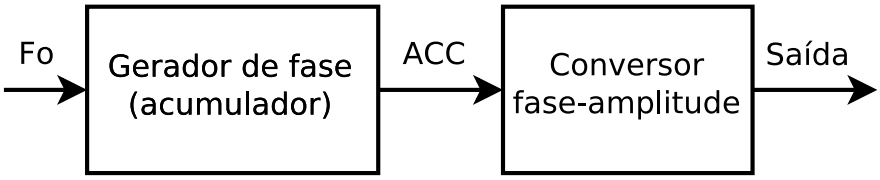

Figura 3.1: Diagrama em Blocos de um DDS

de fase é um acumulador de números inteiros de N bits, em que o valor máximo do acumulador $(2^N - 1)$ corresponde ao valor máximo da fase da onda (2π). Uma LUT (*lookup table*) é conectada à saída do acumulador de fase e faz o papel do conversor fase-amplitude, armazenando os valores da amplitude da forma de onda (seno ou cosseno) relativos à sua fase.

Com essa estrutura, o DDS pode gerar diversos sinais periódicos: senoidais (com os valores contidos na LUT), dente-de-serra (com os valores do acumulador), triangulares (a partir da dente-de-serra) e retangulares (utilizando o bit mais significativo do acumulador). Se um sinal analógico for desejado, pode-se conectar um conversor D/A (digital para analógico) e um filtro passa-baixa na saída do DDS.

Como o acumulador de fase tem 2^N valores únicos, a LUT deve armazenar até 2^N valores de amplitude da onda. Porém, é inviável armazenar todos estes valores, porque o tamanho da memória ROM seria muito grande para uma implementação prática. Para tornar viável a implementação via LUT é necessário realizar um descarte de informação, o que causa o aparecimento de componentes espectrais não desejadas, chamadas de *espúrios*. A diferença entre o nível da portadora (sinal desejado) e o máximo nível de espúrios é chamada de faixa dinâmica livre de espúrios (SFDR) [10].

A fim de diminuir a SFDR sem necessitar mais espaço em memória, foram desenvolvidas técnicas de compressão, para reduzir o espaço ocupado. Explorando a simetria do quarto de onda da função seno, é possível reproduzir a faixa completa da onda (0 a 2π) armazenando apenas a faixa de 0 a $\pi/2$ de informação. Aproximações trigonométricas também podem ser utilizadas para reduzir o espaço ocupado em memória. Entre estas, podemos citar as arquiteturas Sunderland, Sunderland modificada e Ni-

cholas, a aproximação por série de Taylor, entre outras. Para mais informações quanto a este assunto, ver referências [10] e [11].

O modo de rotação do CORDIC pode ser utilizado como mapeador fase-amplitude, gerando diretamente formas de onda do seno e do cosseno. Usando o CORDIC, é possível obter alta resolução de fase e alta precisão com baixo custo de *hardware*. Entretanto, é necessária uma pequena modificação no acumulador de fase para o funcionamento correto do DDS baseado em CORDIC. Em vez de gerar um número inteiro correspondente a um endereço de memória, é preciso que o acumulador gere um ângulo. Este pode ser obtido alterando-se o acumulador para que este tenha apenas um bit inteiro, e introduzindo um multiplicador por um valor aproximado de π na saída do acumulador.

3.2 Sincronização

O componente principal para a sincronização, tanto de fase quanto de freqüência, é o PLL (*Phase Locked Loop*), que é composto por um detector de fase, um oscilador e um filtro passa-baixa, que são conectados em uma configuração realimentada [12].

Destes componentes, os dois primeiros podem ser realizados utilizando CORDIC: o oscilador é substituído por um DDS (com o CORDIC no papel de mapeador fase-amplitude), e a detecção de fase pode ser realizada por um processador CORDIC operando no modo de vetorização. Esta é realizada comparando-se a diferença de fase entre o sinal recebido e o ponto da constelação mais próximo, como é representado na Figura 3.2.

Em constelações complexas (ex.: 16-QAM), tornam-se necessárias comparações adicionais para o cálculo preciso da diferença de fase entre o sinal e a referência. Para isto, um *slicer* de duas dimensões é integrado ao detector de fase, como pode ser visto na Figura 3.3.

Em uma implementação em FPGA, o sistema contendo o detector de fase por CORDIC recupera a portadora em aproximadamente o mesmo tempo de uma implementação por aritmética de ponto flutuante e seis vezes mais rápido do que um detector

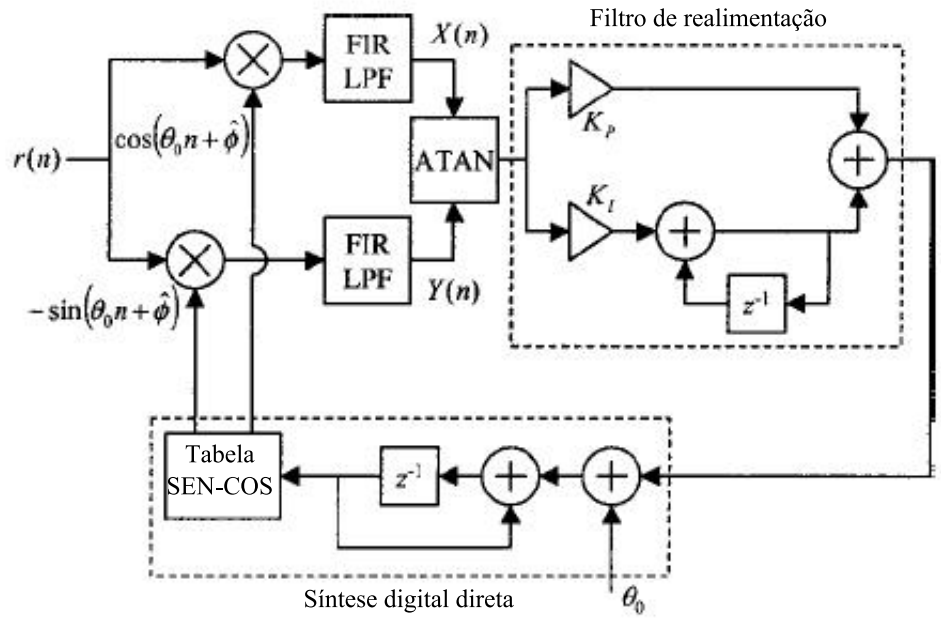

Figura 3.2: PLL utilizado para sincronização de fase em portadora [12]

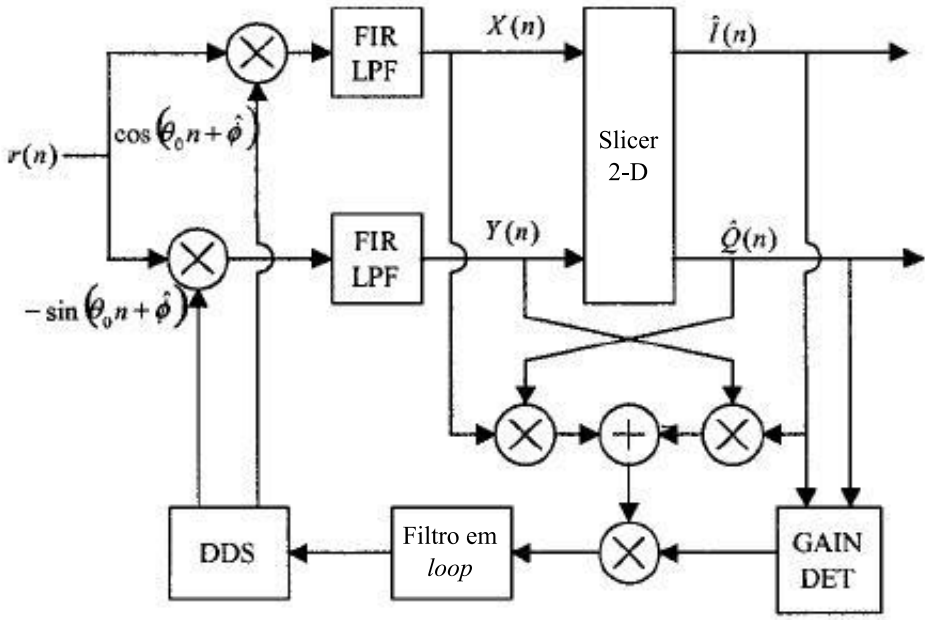

Figura 3.3: PLL utilizado para sincronização de constelações bidimensionais [12]

de fase utilizando uma LUT com tamanho de 4096x8 bits [12].

3.3 *Up/Downconversion*

A conversão digital de uma freqüência mais alta para uma mais baixa ou vice-versa é um processo muito comum em sistemas de comunicação. Tal processo pode ser facilmente desempenhado por um processador CORDIC operando no modo de rotação.

Um *mixer* para *upconversion* pode ser realizado conectando-se os sinais (em banda-base) em fase e quadratura nas entradas X_0 e Y_0, respectivamente, como pode ser observado na Figura 3.4. Com isso, na saída X_N será obtido o sinal $s(n) = K[s_i(n)\cos(f_c\pi n) - s_q(n)sen(f_c\pi n)]$. Como se pode perceber, é necessário um ganho final de $1/K$ para a forma final do sinal.

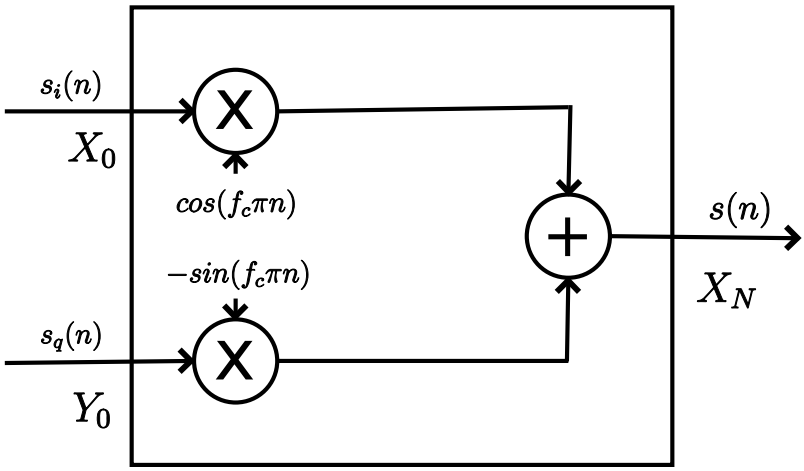

Figura 3.4: Mixer para *upconversion* [8]

Para separar os componentes em fase e quadratura, o sinal recebido $r(n)$ é conectado à entrada Y_0 do processador CORDIC, com X_0 sendo zerado. A Figura 3.5 representa este processo, mostrando assim o sinal obtido na saída X_N, isto é, $r_q(n) = -K \cdot r(n) \cdot sen(f_c\pi n)$, e o sinal na saída Y_N, isto é, o sinal $r_i(n) = K \cdot r(n) \cdot \cos(f_c\pi n)$.

Mas é no caso de multiplexagem de sinais em quadratura que o CORDIC tem maior vantagem em relação ao sistema baseado em LUTs, pois um processador CORDIC operando no modo de rotação substitui quatro multiplicadores e a LUT do DDS [8]. A Figura 3.6 mostra as conexões necessárias para realizar a multiplexagem de sinais em quadratura. Numa conversão para uma freqüência mais alta, o sinal em fase I_1 é conectado à entrada X_0, e o sinal em quadratura Q_1 à entrada Y_0, re-

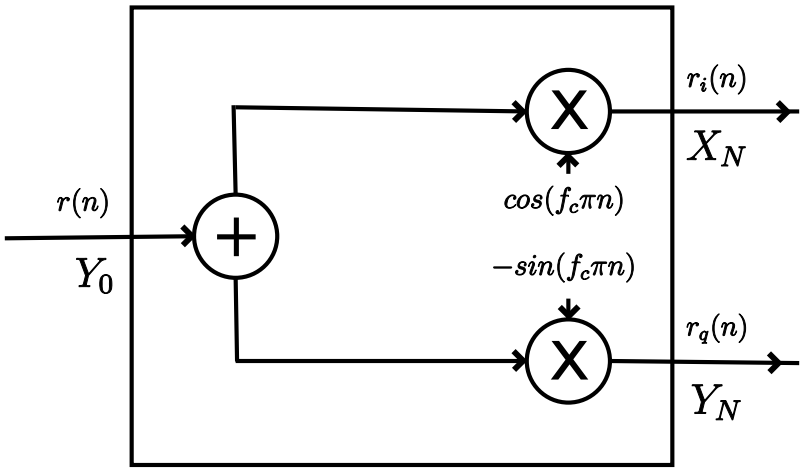

Figura 3.5: Mixer para *downconversion* [8]

sultando nas seguintes saídas: $I_2 = K[I_1 \cos(f_c\pi n) - Q_1 sen(f_c\pi n)]$ na porta X_N e $Q_2 = K[I_1 sen(f_c\pi n) + Q_1 \cos(f_c\pi n)]$ na porta Y_N.

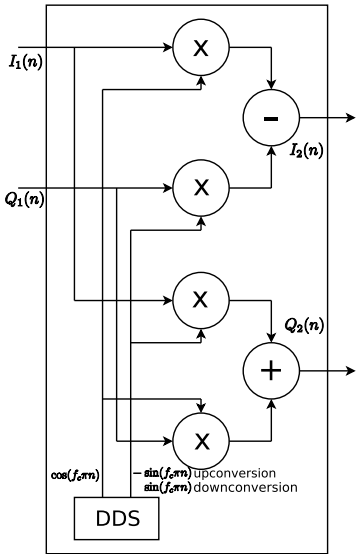

Figura 3.6: Mixer complexo para *up/downconversion* [8]

Já no caso de conversão para uma freqüência mais baixa, o sinal em fase I_1 é conectado à entrada Y_0, e o sinal em quadratura Q_1 à entrada X_0, resultando nas seguintes saídas: $I_2 = K[Q_1 sen(f_c\pi n) + I_1 \cos(f_c\pi n)]$ na porta Y_N e $Q_2 = K[Q_1 \cos(f_c\pi n) - I_1 sen(f_c\pi n)]$ na porta X_N.

3.4 Modulação

O CORDIC pode ser utilizado tanto para a geração das modulações analógicas AM, PM, e FM quanto das digitais ASK, PSK e FSK. A Figura 3.7 mostra as conexões necessárias para a geração dos sinais modulados.

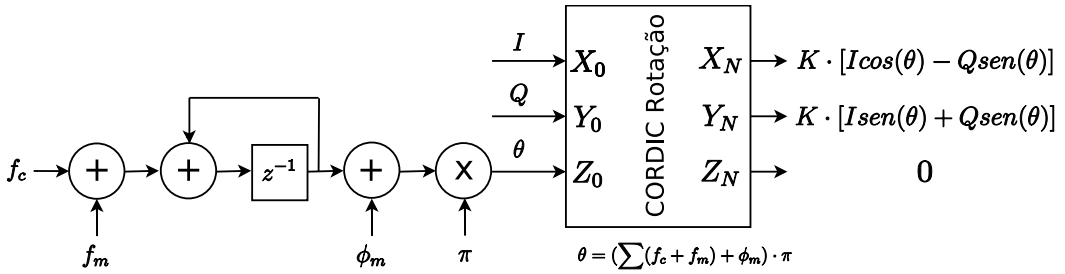

Figura 3.7: Esquema para modulação utilizando CORDIC [8]

Para obter modulação de amplitude (AM/ASK) basta conectar o sinal modulador $m(n)$ na entrada X_0 do processador CORDIC. Assim, é obtida na saída X_N a forma de onda $s(n) = m(n)\cos(f_c\pi n)$.

Se o objetivo for uma modulação de fase (PM/PSK), o sinal $m(n)$ deve ser conectado na entrada ϕ_m, e a entrada X_0 tem o valor fixado em $1/K_N$. Desta forma, o sinal de saída será $s(n) = \cos(f_c\pi n + m(n)\pi)$.

E se uma modulação de freqüência (FM/FSK) for desejada, o sinal modulador deve ser conectado na entrada f_m, com um valor da freqüência da portadora f_c fixo e a entrada X_0 fixada em $1/K_N$. Com isso, na saída X_N temos o sinal $s(n) = \cos(f_c\pi n + (\sum m(n)\pi))$.

Antes do processador CORDIC ser utilizado em um modulador AM, PM ou FM, é necessário realizar um *upsampling* no sinal modulador em banda base $m(n)$ para adequá-lo à freqüência de amostragem do DAC e do processador CORDIC. Para isto, uma solução que não exige muito espaço em *hardware* é um filtro CIC (Cascade-Integrator-Comb) [13]. A Figura 3.8 mostra um filtro CIC em dois modos de operação: como decimador e como interpolador.

O CORDIC também pode ser utilizado para a modulação QAM. Nesta, o processador CORDIC realiza uma rotação circular do vetor $[I(n)Q(n)]^T$, onde $I(n)$ é o

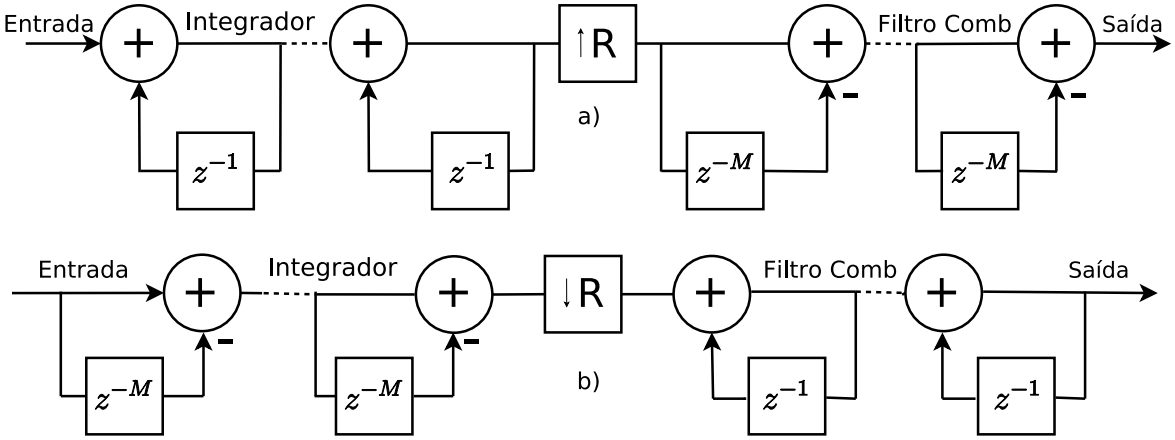

Figura 3.8: Filtro CIC - a) decimador; b) interpolador

componente em fase e $Q(n)$ o componente em quadratura do sinal a ser modulado [14]. Assim, na saída X_N do processador CORDIC, é obtida a saída do modulador QAM:

$$s(n) = I(n)\cos(f_c t(n)) + Q(n)\sin(f_c t(n)), \tag{3.1}$$

onde f_c é a freqüência de saída do DDS.

3.5 DFT e FFT

Dada uma seqüência discreta de números complexos $x(n)$, $0 \leq n \leq N-1$, sua transformada discreta de Fourier (DFT) é definida por:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi nk}{N}}, k = 0, 1, ..., N-1. \tag{3.2}$$

A DFT pode ser entendida como uma rotação do vetor $x(n)$ pelo ângulo $2\pi nk/N$, seguida por uma soma para todo n [15]. Então, a DFT pode ser reescrita da seguinte forma:

$$\begin{bmatrix} X_r(n+1,k) \\ X_i(n+1,k) \end{bmatrix} = K_1(n) \cdot \begin{bmatrix} cos(2\pi nk/N) & -sen(2\pi nk/N) \\ sen(2\pi nk/N) & cos(2\pi nk/N) \end{bmatrix} \cdot \begin{bmatrix} x_r(n) \\ x_i(n) \end{bmatrix} + \begin{bmatrix} X_r(n,k) \\ X_i(n,k) \end{bmatrix},$$
$$k = 0, \ldots, N-1, n = 0, \ldots, N-1 \tag{3.3}$$

onde k é o índice do processador CORDIC, n o estágio da DFT, X_r a parte real da seqüência X_k, X_i corresponde à parte imaginária de X_k e x_r e x_i são os números propagados pelo processador CORDIC anterior da rede.

Assim, a DFT pode ser calculada facilmente utilizando um processador CORDIC. O ganho $K_1(n)$, que é proveniente das rotações calculadas através do CORDIC, pode ser compensado apenas no final da DFT, necessitando assim menos iterações e diminuindo o atraso causado pela DFT. A Figura 3.9 ilustra a DFT utilizando CORDIC.

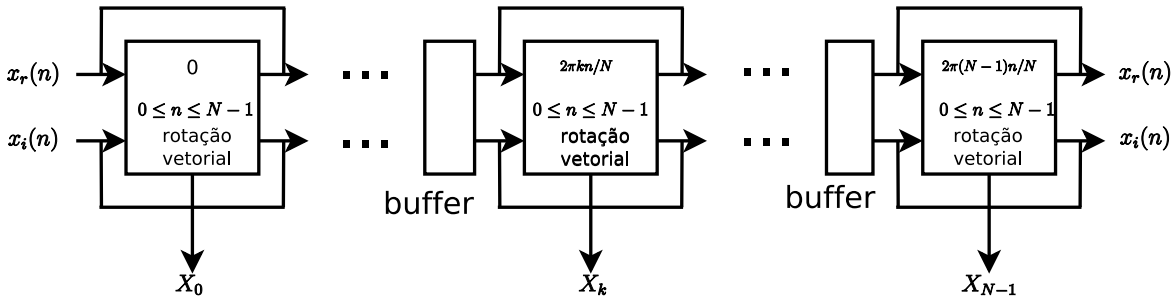

Figura 3.9: DFT utilizando CORDIC

A FFT [16] é uma implementação da DFT que permite sua rápida execução. Para tal, ela transforma uma DFT de N pontos em duas DFTs de $N/2$ pontos recursivamente (no caso do algoritmo radix-2) [17]. A unidade básica de uma FFT é a *butterfly*, que corresponde a uma DFT de 2 pontos.

A operação *butterfly* com decimação no tempo é formulada da seguinte maneira:

$$C' = C + De^{-j(2\pi nk/N)}, \tag{3.4}$$

$$D' = C - De^{-j(2\pi nk/N)}. \tag{3.5}$$

Já a operação *butterfly* com decimação na freqüência é formulada da seguinte maneira:

$$C' = C + D, \tag{3.6}$$

$$D' = (C - D) \cdot e^{-j(2\pi nk/N)}, \tag{3.7}$$

onde C, D, C' e D' são números complexos; C e D representam os dados de entrada da *butterfly* e C' e D' são os resultados na saída da *butterfly*. A Figura 3.10 representa

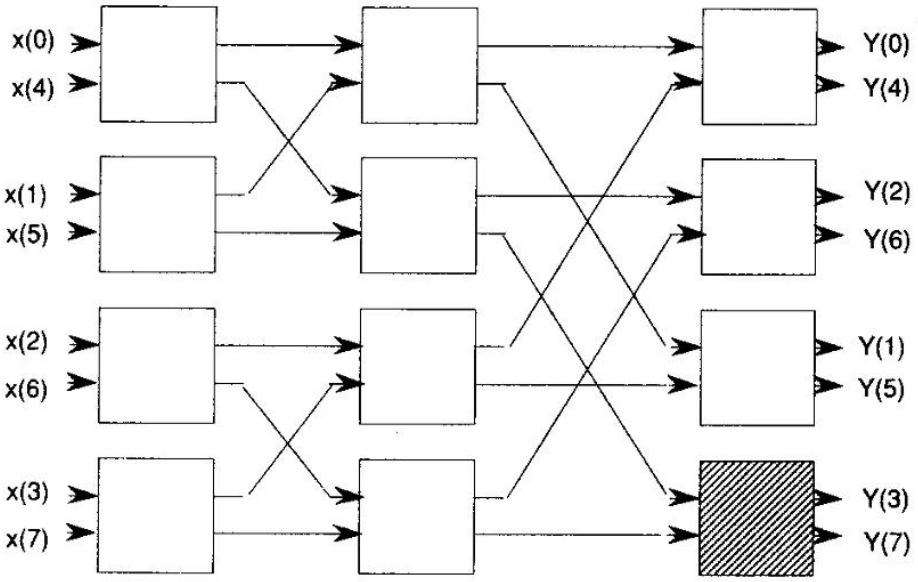

Figura 3.10: FFT utilizando CORDIC [17]

a implementação de uma FFT de 8 pontos via rede de processadores CORDIC. Cada caixa corresponde a uma rotação CORDIC seguida de uma adição/subtração complexa.

Sarmiento [18] propõe um processador utilizando CORDIC para a FFT de 1024 pontos, que requer 18 estágios: inicialização, sete microrotações radix-2, quatro rotações radix-4, quatro repetições de rotações radix-2 e dois estágios de ganho. Como nesta implementação os ângulos de rotação são conhecidos *a priori* e armazenados em uma ROM, a utilização da terceira equação 2.20 do algoritmo CORDIC não se faz necessária. Este processador, baseado em CORDIC e construído utilizando tecnologia de arseneto de gálio (GaAs), calcula uma FFT de 1024 pontos, com dados de entrada complexos de 16 bits, em $8\mu s$, operando a mais de 700 MHz e consumindo 12,5 W.

Garrido [19], através de modificações nas rotações angulares, implementa um processador CORDIC em FPGA para o cálculo da FFT que, com 16 iterações e com dados de entrada de 16 bits, opera a 255 MHz em uma FPGA modelo Xilinx Virtex-II xc2v4000-6.

3.6 Filtragem Digital

Em geral, a maior parte das operações de filtragem é realizada com operações de multiplicação e acumulação (MAC). Entretanto, existem classes de filtros que podem ser implementadas de maneira mais eficiente utilizando CORDIC. Entre elas, podem-se citar dois tipos principais: o *filtro digital ortogonal* (em inglês, Orthogonal Digital Filter - ODF), baseado em rotações circulares, e o *filtro em treliça adaptativo* (em inglês, Adaptive Lattice Filter - ALF), baseado em rotações circulares e hiperbólicas [17].

O filtro digital ortogonal é um filtro que possui uma estrutura numérica a cada amostra que pode ser descrita por uma matriz ortogonal ou unitária. Comparado com outros filtros, ele tem baixa sensibilidade tanto na banda de passagem quanto na banda de corte, e é invariante sob transformação de freqüência. Também é estável mesmo com quantização de parâmetros e livre de oscilações de *overflow*. Além disso, possui uma estrutura modular [17].

A unidade básica de um ODF é um rotor. Este rotor é uma unidade que realiza uma rotação circular, expressa pela seguinte operação matemática:

$$\begin{bmatrix} x \\ g \end{bmatrix} = \begin{bmatrix} \sqrt{1-k^2} & k \\ -k & \sqrt{1-k^2} \end{bmatrix} \cdot \begin{bmatrix} f \\ y \end{bmatrix} = \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} \cdot \begin{bmatrix} f \\ y \end{bmatrix}, \quad (3.8)$$

onde f e y são as entradas e x e g são as saídas. A Figura 3.11 representa as operações matemáticas da Equação 3.8.

A principal aplicação de um filtro digital ortogonal é a implementação de funções de transferência com coeficientes fixos. Mais informações sobre o filtro digital ortogonal podem ser encontradas em [17].

Já o filtro em treliça adaptativo é mais apropriado para o processamento adaptativo de sinais, já que os filtros em treliça mais populares são focados na atualização adaptativa dos coeficientes de reflexão, para que seja possível o ajuste frente aos parâmetros variáveis do sinal de entrada. A unidade básica do filtro em treliça adaptativo é baseada em rotações hiperbólicas, que podem ser realizadas por um processador

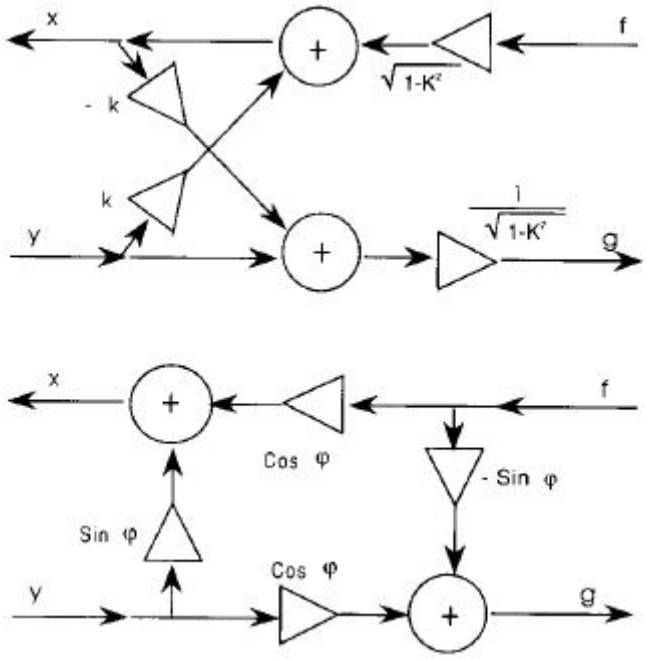

Figura 3.11: Rotor ODF [17]

CORDIC operando em modo hiperbólico. Esta rotação pode ser descrita por:

$$\begin{bmatrix} f \\ g \end{bmatrix} = \frac{1}{\sqrt{1-k^2}} \begin{bmatrix} 1 & -k \\ -k & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cosh(\theta) & -\sinh(\theta) \\ -\sinh(\theta) & \cosh(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \quad (3.9)$$

onde k é o coeficiente de reflexão ou correlação parcial. Para manter a estabilidade do filtro, $|k|$ deve ser menor do que 1. A Figura 3.12 ilustra a unidade que realiza a Equação 3.9.

Existem duas formas para adaptar $k_i(t)$ a cada instante de tempo: os filtros de gradiente em treliça (GLF) e os filtros em treliça de mínimos quadrados recursivos (RLSLF). Os primeiros utilizam formulações baseadas em gradiente para a adaptação dos coeficientes, enquanto os últimos resolvem um problema de mínimos quadrados para realizar a adaptação. A estrutura destes filtros é facilmente implementada em CORDIC, mas a adaptação dos coeficientes de reflexão torna-se mais difícil de ser implementada [17].

Para contornar este problema, Hu e Liao propuseram um filtro adaptativo em treliça que utiliza processadores CORDIC [2]. Tal filtro é baseado em uma adaptação simplificada por gradiente, e atualiza diretamente o ângulo de rotação hiperbólico, em vez de atualizar os coeficientes de reflexão. Um filtro de predição linear em treliça de

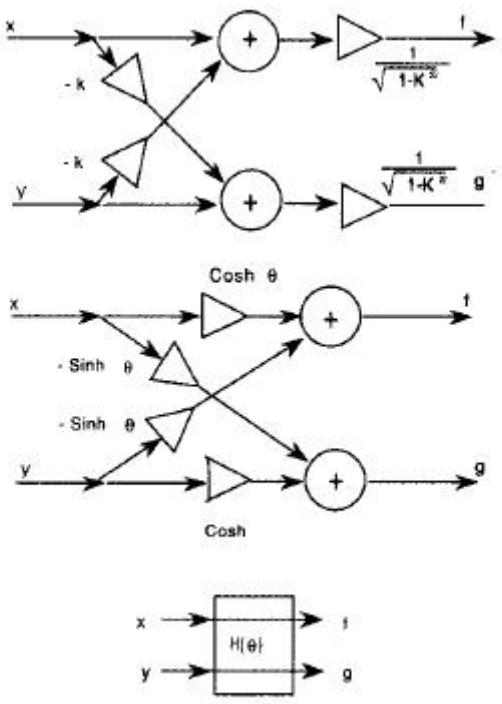

Figura 3.12: Unidade básica de um filtro ALF [17]

ordem n no estágio i da treliça no tempo t utilizando o CALF (em inglês, CORDIC Adaptive Lattice Filter - Filtro Adaptivo em Treliça utilizando CORDIC) pode ser formulado da seguinte maneira:

$$\begin{bmatrix} \overline{f}_i(t) \\ \overline{b}_i(t) \end{bmatrix} = \begin{bmatrix} \cosh(\theta) & -\sinh(\theta) \\ -\sinh(\theta) & \cosh(\theta) \end{bmatrix} \begin{bmatrix} \overline{f}_{i-1}(t) \\ \overline{b}_{i-1}(t-1) \end{bmatrix}, \tag{3.10}$$

onde $\overline{f}_i(t)$ e $\overline{b}_i(t)$ são os erros de predição *forward* e *backward*, respectivamente. O ângulo de rotação hiperbólico é atualizado através da seguinte fórmula:

$$\theta_i(t) = \theta_i(t-1) + \mu_i \overline{f}_i(t-1)\overline{b}_i(t-1). \tag{3.11}$$

A Figura 3.13 representa uma unidade básica para o cálculo das equações que compõe o CALF.

O CALF geralmente é utilizado para a implementação de filtros AR (autoregressivo) em treliça. Entretanto, é difícil aplicar o algoritmo CALF para um filtro ARMA (em inglês, *autoregressive moving average* - média móvel autoregressiva) em treliça normalizado, devido aos problemas de escolha entre alta precisão e alta velocidade de convergência e o ângulo de rotação convergindo à metade do ângulo de rotação

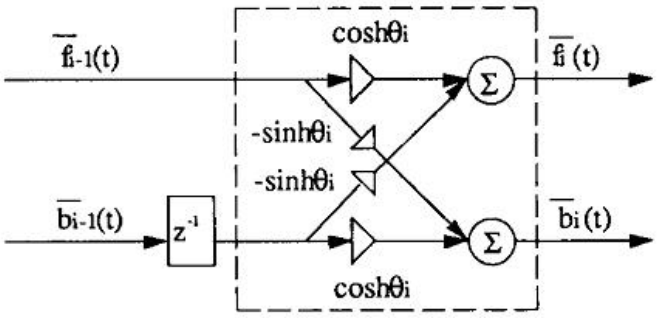

Figura 3.13: Unidade básica de um filtro CALF [2]

verdadeiro calculado pela fórmula de Burg [20] nos ângulos de rotação calculados. Por isso, Shiraishi *et al* [3] propuseram uma nova estrutura baseada em CORDIC para um filtro ARMA em treliça. Esta nova estrutura também possibilita que o o filtro ARMA em treliça possa, além de funcionar como filtro de análise, servir como um filtro de síntese de sinais, mantendo a estrutura sem multiplicadores.

3.7 Conclusão

Neste capítulo, foram introduzidas algumas das várias aplicações possíveis do algoritmo CORDIC. Entre elas: a síntese digital direta (DDS), sincronização, modulação, *up/downconversion*, FFT e filtragem digital. Estas aplicações mostram a grande flexibilidade do CORDIC, que torna possível realizar várias operações diferentes utilizando o mesmo *hardware*. O próximo capítulo apresentará um equalizador adaptativo baseado no algoritmo LMS trigonométrico, que utiliza processadores CORDIC para as operações de filtragem e atualização dos coeficientes. O algoritmo LMS trigonométrico depende da definição de um hipercubo no qual esteja contido o mínimo da função objetivo. Uma forma de contornar a necessidade da definição desse hipercubo é a utilização da técnica multisplit para a implementação do equalizador.

Capítulo 4

Equalização Adaptativa Trigonométrica Multisplit

Em grande parte dos sistemas de comunicação digital, os canais são variantes no tempo. Neles também ocorre uma dispersão temporal do sinal transmitido, fazendo com que dados transmitidos em um instante interfiram com dados transmitidos em outros instantes. Este fenômeno é conhecido como interferência entre símbolos (IES). A IES provoca a redução da taxa de dados transmitidos e/ou o aumento da probabilidade de erro no canal.

A redução dos efeitos da IES torna-se necessária quando a duração T_s do símbolo é da mesma ordem de grandeza do valor rms T_σ do espalhamento de atraso do canal. Uma maneira para a compensação dos efeitos da IES é a utilização de um equalizador. A função de um equalizador é compensar o efeito do canal e produzir uma estimativa correta do símbolo recebido [21].

A Figura 4.1 ilustra uma constelação QPSK (item a), os efeitos de um canal dispersivo nesta constelação (item b) e a constelação equalizada (item c). Como é possível ver, o diagrama de olho após o equalizador está bem aberto, facilitando assim a decisão correta dos símbolos.

Durante o projeto de um equalizador, um ponto importante é o equilíbrio entre a redução da IES e a amplificação do ruído, já que tanto o sinal quanto o ruído passam pelo equalizador. Além disso, como os canais são variantes no tempo, é necessário que

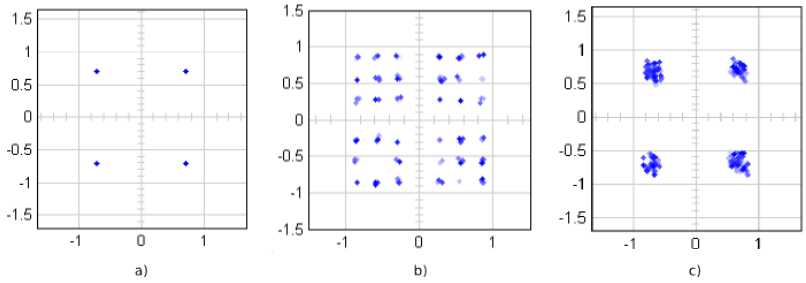

Figura 4.1: Efeitos de um canal sobre uma constelação QPSK e sua equalização - a) constelação QPSK - b) efeito do canal dispersivo - c) constelação equalizada

o equalizador seja adaptativo. Por isso, deve-se usar um algoritmo que permita ajustar os valores dos parâmetros. Tal adaptação é realizada utilizando um sinal de erro para este processo de ajuste. O sinal de erro pode ser calculado através da diferença entre um sinal de referência (o sinal desejado) e a saída do equalizador.

Os equalizadores podem ser implementados em banda base, RF ou IF. A maioria deles é implementada digitalmente, depois de um processo de conversão A/D [21].

Anteriormente a [4], filtros adaptativos baseados em CORDIC utilizando o algoritmo LMS para adaptação dos coeficientes apenas eram apresentados em forma de treliça, já que nestes, cada estágio pode ser facilmente relacionado a um conjunto de rotações hiperbólicas, ao contrário dos filtros transversais. Em [4], uma realização alternativa do algoritmo LMS é proposta, utilizando variáveis trigonométricas, que são relacionadas monotonicamente aos coeficientes dos filtros. O algoritmo atualiza os ângulos, e a partir destes são gerados os coeficientes. Este algoritmo é especialmente apropriado para ser realizado com um processador CORDIC, já que este calcula simultaneamente as funções trigonométricas necessárias para os processos de filtragem e atualização dos coeficientes. Assim, metade dos multiplicadores podem ser substituídos por processadores CORDIC.

Este capítulo apresenta um equalizador adaptativo que utiliza o algoritmo LMS trigonométrico para ajustar seus coeficientes. A função objetivo a ser minimizada é a potência do erro entre o sinal desejado e a saída do equalizador. A utilização deste algoritmo impõe a necessidade de se definir um hipercubo, no qual deve estar contido o ponto de mínimo da função objetivo. Os artigos disponíveis na literatura não são

claros quanto à definição da dimensão deste hipercubo. Nessa dissertação é proposta a utilização da técnica multisplit como uma maneira de fixar as coordenadas do hipercubo para um grande número de canais. Como a técnica multisplit será implementada via transformada de Hadamard, será apresentada uma breve introdução teórica desta transformada.

4.1 O algoritmo LMS trigonométrico

Para analisar a formulação matemática do algoritmo LMS trigonométrico, considera-se o procedimento de "*steepest descent*"de filtragem FIR ótima. Assim, é necessário definir:

- a seqüência de entrada:

$$\mathbf{x}(n) = [x(n), x(n-1), \ldots, x(n-N+1)]^T, \tag{4.1}$$

- N números positivos A_k, $k = 0, 1, \ldots, N-1$,
- o vetor de coeficientes do filtro:

$$\mathbf{w}(n) = [w(0), w(1), \ldots, w(N-1)]^T, \tag{4.2}$$

- e a resposta desejada $d(n)$.

Primeiramente, são escolhidos N números positivos A_k, $k = 0, 1, \ldots, N-1$, para a definição do hipercubo que contém os mínimos da superfície de desempenho de erro. Tal hipercubo terá os vértices $[\pm A_0, \pm A_1, \ldots \pm A_{N-1}]$. Desta maneira, pode-se expressar os coeficientes w_k do filtro como:

$$w_k = A_k \sin\theta_k, k = 0, 1, \ldots, N-1, \tag{4.3}$$

com $-\pi/2 < \theta < \pi/2$.

Como cada $w_k \in (-A_k, +A_k)$ e os valores A_k são fixos, os coeficientes w_k dependem unicamente dos ângulos θ_k. Desta maneira, a variável a ser efetivamente adaptada é θ_k.

Define-se o filtro ótimo de Wiener $\hat{\mathbf{w}} = [\hat{w}(0), \hat{w}(1), \ldots, \hat{w}(N-1)]^T$ pela minimização da função de erro $\epsilon^2 = E[e(n)e^*(n)]$, onde $e(n) = d(n) - \mathbf{w}^T\mathbf{x}(n)$. O erro médio quadrático ϵ^2 é função dos coeficientes do filtro $\mathbf{w}$, e define a superfície de desempenho de erro de dimensão $N+1$.

Uma busca utilizando o algoritmo "*steepest descent*"é feita dentro do hipercubo no espaço θ para alcançar o $\widehat{\theta}$ ótimo. Utilizando-se procedimentos matemáticos relativamente simples, o vetor gradiente

$$\nabla_\theta \epsilon^2 = [\partial\epsilon^2/\partial\theta_0, \partial\epsilon^2/\partial\theta_1, \ldots, \partial\epsilon^2/\partial\theta_{N-1}]^t \tag{4.4}$$

pode ser escrito como:

$$\nabla_\theta \epsilon^2 = -2\Delta(\mathbf{p} - \mathbf{R}\mathbf{w}), \tag{4.5}$$

onde:

$$\mathbf{w} = [A_0 \sin\theta_0, A_1 \sin\theta_1, \ldots\ A_{N-1}\sin\theta_{N-1}]^t, \tag{4.6}$$

$$\mathbf{p} = E[\mathbf{x}(n)d(n)], \tag{4.7}$$

$$\mathbf{R} = E[\mathbf{x}(n)\mathbf{x}^t(n)], \tag{4.8}$$

e Δ é uma matriz diagonal com o j-ésimo elemento dado por:

$$\delta_{j,j} = A_j \cos\theta_j, j = 0, 1, \ldots, N-1. \tag{4.9}$$

Assim, a iteração $\theta(i)$ é calculada como:

$$\theta(i+1) = \theta(i) - (\mu/2)\nabla_\theta \epsilon^2|_{\theta=\theta(i)}, \tag{4.10}$$

onde μ é o passo de adaptação do algoritmo.

Para passar do modo "*steepest descent*"para a forma LMS, é necessário substituir as esperanças na Equação 4.5 por seus valores instantâneos, a fim de obter uma estimativa do gradiente no instante n. Desta forma, chega-se ao LMS trigonométrico, cujas equações são expostas a seguir:

$$e(n) = d(n) - \sum_{k=0}^{N-1} A_k \sin\theta_k(n)x(n-k). \tag{4.11}$$

$$\theta(n+1) = \theta(n) + \mu\Delta(n)\mathbf{x}(n)e(n), \tag{4.12}$$

O algoritmo LMS trigonométrico é especialmente apropriado para a realização via processadores CORDIC. Isto se deve ao fato de que é possível calcular simultaneamente as funções trigonométricas, necessárias tanto para a filtragem quanto para a adaptação dos coeficientes, utilizando apenas um processador CORDIC.

Para uma realização *pipelined* do LMS trigonométrico, é mais apropriado considerar o LMS atrasado (DLMS) [22, 23]. Neste, os coeficientes no instante n são atualizados utilizando uma estimativa anterior do gradiente (por exemplo, um instante $(n - D)$, onde D é um inteiro). Assim, a equação de atualização dos coeficientes do LMS trigonométrico atrasado é dada por:

$$\theta(n+1) = \theta(n) + \mu\Delta(n-D)\mathbf{x}(n-D)e(n-D). \tag{4.13}$$

Esta variação do LMS necessita de um valor de passo menor para convergência, mas possibilita a adoção de uma taxa de amostragem muito maior [24].

A análise de convergência do LMS trigonométrico é extremamente complexa, devido à não-linearidade das funções trigonométricas utilizadas pelo algoritmo. Para mais detalhes, ver [4].

O algoritmo LMS, tanto em sua versão convencional quanto em sua versão trigonométrica, tem sua taxa de convergência afetada pelo espalhamento dos autovalores da matriz de correlação do sinal de entrada [25]. O algoritmo trigonométrico ainda apresenta o problema de definição do hipercubo para várias aplicações diferentes. A utilização da estrutura multisplit proposta em [26] é uma maneira para resolver estes problemas com baixo custo computacional, pois não requer multiplicações.

4.2 O Equalizador Adaptativo Trigonométrico Multisplit

Qualquer seqüência finita pode ser expressa pela soma de suas partes simétricas e anti-simétricas. Portanto, qualquer filtro FIR pode ser implementado conforme mostrado na Figura 4.2, onde $\mathbf{w}_s$ é a parte simétrica do filtro, enquanto $\mathbf{w}_a$ é a parte anti-simétrica.

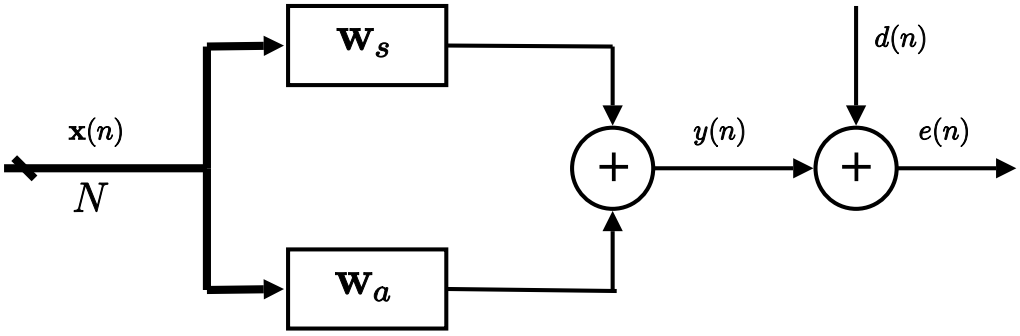

Figura 4.2: Divisão de uma seqüência em suas partes simétricas e anti-simétricas

Com o vetor de coeficientes do filtro FIR dado por

$$\mathbf{w} = [w_0, w_1, \ldots, w_{N-1}]^t, \tag{4.14}$$

pode-se mostrar que [26]:

$$\mathbf{w}_s = (\mathbf{w} + \mathbf{J}_N\mathbf{w})/2 \tag{4.15}$$

e

$$\mathbf{w}_a = (\mathbf{w} - \mathbf{J}_N\mathbf{w})/2, \tag{4.16}$$

onde $\mathbf{J}_N$ é a matrix de reflexão NxN, que pode ser expressa da seguinte maneira:

$$\mathbf{J}_N = \begin{bmatrix} 0 & 0 & \ldots & 1 \\ 0 & \ldots & 1 & 0 \\ 0 & \ddots & 0 & 0 \\ 1 & \ldots & 0 & 0 \end{bmatrix}. \tag{4.17}$$

As condições de simetria e anti-simetria das seqüências $\mathbf{w}_s$ e $\mathbf{w}_a$ são expressas, respectivamente, como:

$$\mathbf{w}_s = \mathbf{J}_N\mathbf{w}_s \tag{4.18}$$

e

$$\mathbf{w}_a = -\mathbf{J}_N\mathbf{w}_a. \tag{4.19}$$

Estas equações podem ser facilmente obtidas a partir das equações 4.15 e 4.16.

As condições de simetria e anti-simetria dos filtros $\mathbf{w}_s$ e $\mathbf{w}_s$ podem ser impostas via restrições lineares. Tais restrições podem ser expressas como:

$$\mathbf{C}_s^t\mathbf{w}_s = \mathbf{f}_s \tag{4.20}$$

e

$$\mathbf{C}_a^t \mathbf{w}_a = \mathbf{f}_a, \tag{4.21}$$

onde $\mathbf{C}_s$ é a matriz de restrição de simetria de dimensão $Nx(N-K)$, definida por:

$$\mathbf{C}_s = \begin{bmatrix} \mathbf{I}_K \\ \\ -\mathbf{J}_K \end{bmatrix}, \tag{4.22}$$

onde $\mathbf{I}_k$ é a matriz identidade de tamanho K e $\mathbf{C}_a$ é a matriz de restrição de anti-simetria de dimensão Nx$(N-K)$, que é definida por:

$$\mathbf{C}_a = \begin{bmatrix} \mathbf{I}_K \\ \\ \mathbf{J}_K \end{bmatrix}. \tag{4.23}$$

Os vetores de resposta $\mathbf{f}_s$ e $\mathbf{f}_a$, que devem ser nulos a fim de satisfazerem as restrições, têm dimensão Kx1.

Como $\mathbf{f}_s = 0$, isso força $\mathbf{w}_s$ a ser ortogonal ao subespaço definido pelas colunas de $\mathbf{C}_s$ [26]. Da mesma forma, $\mathbf{w}_a$ é ortogonal ao subespaço definido pelas colunas de $\mathbf{C}_a$.

Além disso, é fácil de verificar que:

$$\mathbf{C}_s^t \mathbf{C}_a = 0 \tag{4.24}$$

e

$$\mathbf{C}_a^t \mathbf{C}_s = 0. \tag{4.25}$$

O problema agora passa a ser minimizar, com estas duas restrições, a potência do erro. Uma maneira de transformar este problema de minimização restrito em um problema de minimização irrestrito pode ser obtida através da utilização do cancelador de lóbulo lateral (em inglês, *Generalized Sidelobe Canceller* - GSC) [27].

Como as colunas de $\mathbf{C}_a$ e $\mathbf{C}_s$ juntas definem o espaço inteiro $\Re^N$, é possível expressar o vetor de coeficientes $\mathbf{w}_s$ em função desse conjunto de vetores bases como:

$$\mathbf{w}_s = \mathbf{C}_s \mathbf{w}_c + \mathbf{C}_a \mathbf{w}_{\perp s}. \tag{4.26}$$

Os vetores $\mathbf{w}_c$ (de dimensão $Kx1$) e $\mathbf{w}_{\perp s}$ (de dimensão $(N-K)x1$) representam as coordenadas das componentes de $\mathbf{w}$ nos subespaços definidos pelas colunas de C_s e

C_a, respectivamente [28]. Multiplicando-se a Equação 4.26 por $\mathbf{C}_s$, obtém-se:

$$\mathbf{C}_s^t \mathbf{w}_s = \mathbf{C}_s^t \mathbf{C}_s \mathbf{w}_c + \mathbf{C}_s^t \mathbf{C}_a \mathbf{w}_{\perp s}. \tag{4.27}$$

O segundo termo da soma é nulo. Por este motivo, a equação acima pode ser simplificada para:

$$\mathbf{C}_s^t \mathbf{w}_s = \mathbf{C}_s^t \mathbf{C}_s \mathbf{w}_c, \tag{4.28}$$

e o vetor $\mathbf{w}_c$ pode ser definido, utilizando a equação anterior:

$$\mathbf{w}_c = (\mathbf{C}_s^t \mathbf{C}_s)^{-1} \mathbf{f}_s. \tag{4.29}$$

Reescrevendo $\mathbf{w}_s$, obtém-se:

$$\mathbf{w}_s = \mathbf{C}_s (\mathbf{C}_s^t \mathbf{C}_s)^{-1} \mathbf{f}_s + \mathbf{C}_a \mathbf{w}_{\perp s}. \tag{4.30}$$

Como $\mathbf{f}_s = 0$,

$$\mathbf{w}_s = \mathbf{C}_a \mathbf{w}_{\perp s}, \tag{4.31}$$

onde $\mathbf{w}_s$ é um filtro sem restrições de dimensão $(N - K)$x1. A saída desse filtro é:

$$\mathbf{y}_{\perp s} = \mathbf{C}_a^t \mathbf{w}_{\perp s}^t \mathbf{x}(n). \tag{4.32}$$

Este processo pode ser repetido para a parte anti-simétrica. Assim, segundo [26], $\mathbf{y}_{\perp s}$ e $\mathbf{y}_{\perp a}$ são estatisticamente descorrelacionados ($E\mathbf{y}_{\perp s}\mathbf{y}_{\perp a} = 0$). Por este motivo, as partes simétricas e anti-simétricas podem ser otimizadas separadamente. As soluções ótimas são dadas por:

$$\mathbf{w}_{\perp s}^{opt} = (\mathbf{C}_a^t \mathbf{R} \mathbf{C}_a)^{-1} \mathbf{C}_a^t \mathbf{p} \tag{4.33}$$

e

$$\mathbf{w}_{\perp a}^{opt} = (\mathbf{C}_s^t \mathbf{R} \mathbf{C}_s)^{-1} \mathbf{C}_s^t \mathbf{p}. \tag{4.34}$$

Assim, o esquema da Figura 4.3 corresponde às seguintes equações referentes ao filtro *split* ótimo de Wiener:

$$\mathbf{w}_{Wiener}^{opt} = \mathbf{w}_s^{opt} + \mathbf{w}_a^{opt}, \tag{4.35}$$

onde

$$\mathbf{w}_s^{opt} = \mathbf{C}_a \mathbf{w}_{\perp s}^{opt} \tag{4.36}$$

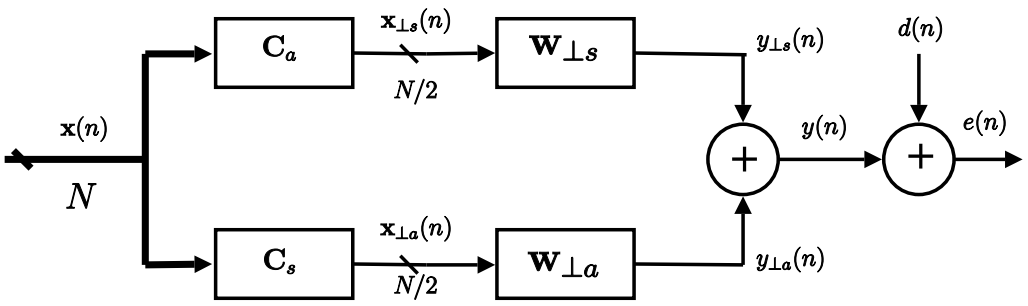

Figura 4.3: Implementação GSC do filtro split

e

$$\mathbf{w}_a^{opt} = \mathbf{C}_s \mathbf{w}_{\perp a}^{opt}. \tag{4.37}$$

Pode-se repetir o processo de *splitting* dos filtros $\mathbf{w}_{\perp s}$ e $\mathbf{w}_{\perp a}$ em suas partes simétricas e anti-simétricas, seguindo os mesmos conceitos anteriores. Depois de L passos, que consistem em 2^{l-1} $(l = 1, 2, \ldots, L)$ operações de *splitting* cada, chega-se a um conjunto de filtros de ordem zero. Esta estrutura está representada na Figura 4.4.

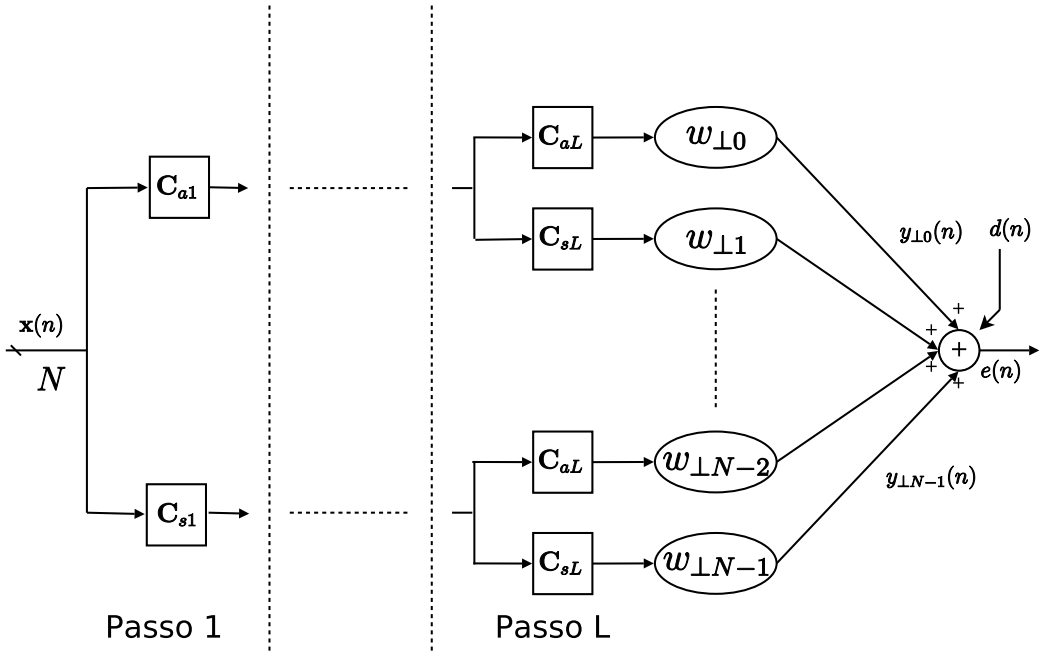

Figura 4.4: Filtragem Adaptativa Multisplit

Desta maneira, o equalizador adaptativo trigonométrico pode ser implementado através de um conjunto de filtros paralelos de um único coeficiente, onde a seqüência

de entrada de cada filtro é dada por:

$$\mathbf{x}_{\perp}(n) = \mathbf{T}^t \mathbf{x}(n), \tag{4.38}$$

onde

$$\mathbf{T} = \begin{bmatrix} \mathbf{C}_{aL}^t & C_{aL-1}^t & \cdots & \mathbf{C}_{a1}^t \\ \mathbf{C}_{sL}^t & \mathbf{C}_{sL-1}^t & \cdots & \mathbf{C}_{s1}^t \\ \vdots & & & \\ \mathbf{C}_{sL}^t & \mathbf{C}_{sL-1}^t & \cdots & \mathbf{C}_{s1}^t \end{bmatrix}_{NxN}^{t} \tag{4.39}$$

e

$$\mathbf{x}_{\perp}(n) = [x_{\perp 0}(n), x_{\perp 1}(n), \ldots, x_{\perp N-1}(n)]^t. \tag{4.40}$$

Para $N = 2^L$, $\mathbf{T}$ é uma matriz composta por +1s e -1s, em que o produto interno entre quaisquer duas colunas é zero, ou seja, as colunas de $\mathbf{T}$ são mutuamente ortogonais. Por este motivo, as colunas de $\mathbf{T}$ podem ser permutadas entre si sem afetar as características da matriz. Uma dessas permutações transforma a matriz $\mathbf{T}$ na matriz de Hadamard de ordem N, possibilitando a representação do equalizador adaptativo trigonométrico multisplit, como mostrado na Figura 4.5.

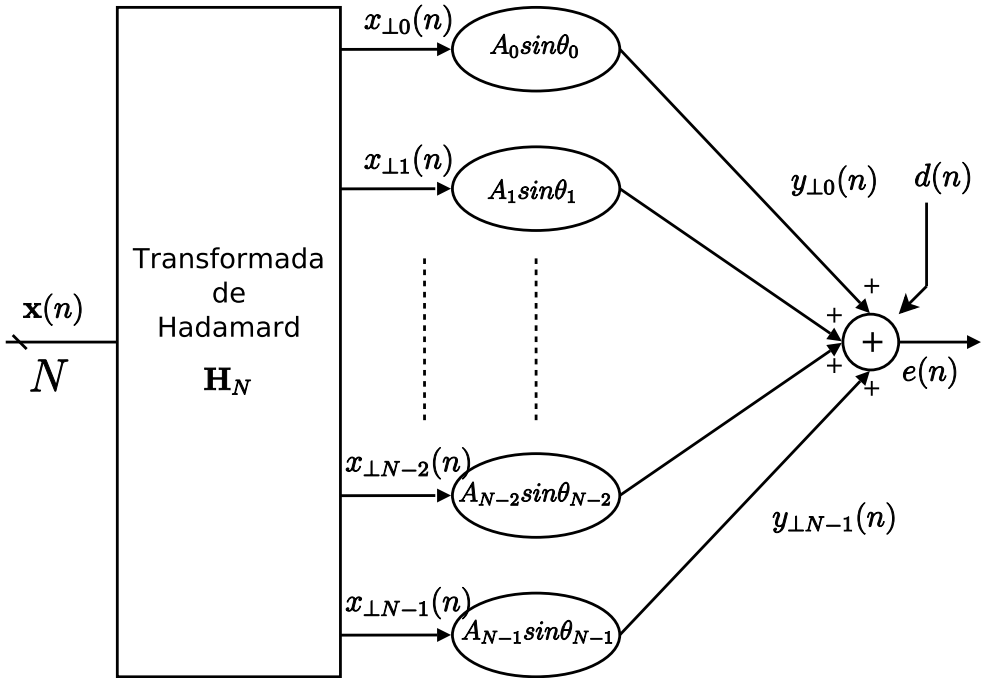

Figura 4.5: Equalizador Adaptativo Trigonométrico Multisplit

A transformada linear realizada no vetor $\mathbf{x}$(n) não o converte em um vetor de variáveis descorrelacionadas, nem afeta o espalhamento dos autovalores da matriz de autocorrelação dos dados de entrada $\mathbf{R}$. Sua principal vantagem é aumentar a diagonalização da matriz de autocorrelação $\mathbf{R}$, levando assim a uma convergência mais rápida na equalização [26].

4.3 Transformada de Hadamard

A transformada de Hadamard é muito utilizada em sistemas de processamento digital de sinais devido à sua fácil implementação e baixo custo computacional. A matriz utilizada pela transformada de Hadamard é composta por ± 1, e tem dimensão de N x N, sendo $N = 2^L$. Uma matriz de Hadamard de ordem N pode ser formada a partir de uma de ordem $N/2$, seguindo o seguinte esquema:

$$\mathbf{H}_{2^L} = \begin{bmatrix} \mathbf{H}_{2^{L-1}} & \mathbf{H}_{2^{L-1}} \\ \mathbf{H}_{2^{L-1}} & -\mathbf{H}_{2^{L-1}} \end{bmatrix}, L = 2, \ldots, \text{ sendo } H_{2^0} = 1. \tag{4.41}$$

Uma outra maneira de descrever esta equação é:

$$\mathbf{H}_{2^l} = \mathbf{H}_2 \otimes H_{2^{l-1}}, \text{ para } l \geq 2, \tag{4.42}$$

onde $\otimes$ denota o produto tensor ou de Kronecker entre matrizes, e

$$\mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{4.43}$$

Todas as colunas ou linhas de uma matriz de Hadamard são mutuamente ortogonais, isto é, o somatório da multiplicação de uma coluna ou linha por outra coluna ou linha qualquer será sempre zero, seguindo a regra abaixo:

$$\sum_{i=0}^{M-1} \mathbf{H}_{ia} \times \mathbf{H}_{ib} = 0, \forall a \neq b. \tag{4.44}$$

4.3.1 Transformada Rápida de Hadamard

A Transformada Rápida de Hadamard (em inglês, Fast Hadamard Transform - FHT) pode ser deduzida a partir da expressão mais simples da transformada de Hadamard:

$$\mathbf{X} = \mathbf{H}\mathbf{x}, \tag{4.45}$$

onde

$$\mathbf{X} = [X(0), X(1), \ldots, X(N-1)]^T \tag{4.46}$$

é o vetor de espectro e

$$\mathbf{x} = [x(0), x(1), \ldots, x(N-1)]^T \tag{4.47}$$

é o vetor correspondente ao sinal de entrada.

Para mostrar a FHT, consideremos $N = 8$. Assim, é possível definir a transformada de Hadamard do vetor $\mathbf{x}$ como:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} = \begin{bmatrix} \mathbf{H}_4 & \mathbf{H}_4 \\ \mathbf{H}_4 & -\mathbf{H}_4 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix}. \tag{4.48}$$

Separando a equação anterior em duas partes, é obtida a primeira parte do vetor $\mathbf{X}(k)$:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \mathbf{H}_4 \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix} + \mathbf{H}_4 \begin{bmatrix} x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix} = \mathbf{H}_4 \begin{bmatrix} x_1(0) \\ x_1(1) \\ x_1(2) \\ x_1(3) \end{bmatrix}, \tag{4.49}$$

onde:

$$x_1(i) = x(i) + x(i+4), i = 0, \ldots, 3. \tag{4.50}$$

A segunda parte do vetor $\mathbf{X}(k)$ pode ser expressa como:

$$\begin{bmatrix} X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} = \mathbf{H}_4 \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix} - \mathbf{H}_4 \begin{bmatrix} x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix} = \mathbf{H}_4 \begin{bmatrix} x_1(4) \\ x_1(5) \\ x_1(6) \\ x_1(7) \end{bmatrix}, \tag{4.51}$$

onde:

$$x_1(i+4) = x(i) - x(i+4), i = 0, \ldots, 3. \tag{4.52}$$

Assim, a transformada de Hadamard de tamanho $N = 8$ foi dividida em duas de tamanho $N/2 = 4$. Continuando as divisões recursivamente, a Equação 4.49 pode ser expressa como:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} \mathbf{H}_2 & \mathbf{H}_2 \\ \mathbf{H}_2 & -\mathbf{H}_2 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \tag{4.53}$$

Tal equação pode ser dividida em duas. A primeira parte é:

$$\begin{bmatrix} X(0) \\ X(1) \end{bmatrix} = \mathbf{H}_2 \begin{bmatrix} x_1(0) \\ x_1(1) \end{bmatrix} + \mathbf{H}_2 \begin{bmatrix} x_1(2) \\ x_1(3) \end{bmatrix} = \mathbf{H}_2 \begin{bmatrix} x_2(0) \\ x_2(1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_2(0) \\ x_2(1) \end{bmatrix}, \tag{4.54}$$

onde:

$$x_2(i) = x_1(i) + x_1(i+2), i = 0, 1. \tag{4.55}$$

A segunda parte da Equação 4.53 consiste em:

$$\begin{bmatrix} X(2) \\ X(3) \end{bmatrix} = \mathbf{H}_2 \begin{bmatrix} x_1(0) \\ x_1(1) \end{bmatrix} - \mathbf{H}_2 \begin{bmatrix} x_1(2) \\ x_1(3) \end{bmatrix} = \mathbf{H}_2 \begin{bmatrix} x_2(2) \\ x_2(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_2(2) \\ x_2(3) \end{bmatrix}, \tag{4.56}$$

onde:

$$x_2(i+2) = x_1(i) - x_1(i+2), i = 0, 1. \tag{4.57}$$

A partir da Equação 4.54 chega-se às equações correspondentes ao resultado da transformada de Hadamard:

$$X(0) = x_2(0) + x_2(1), \tag{4.58}$$

$$X(1) = x_2(0) - x_2(1). \tag{4.59}$$

Com as equações acima, apenas os dois primeiros resultados são obtidos. É necessário aplicar o procedimento acima para os outros dados a fim de obter-se a seqüência completa.

Este procedimento matemático pode ser visualizado na Figura 4.6.

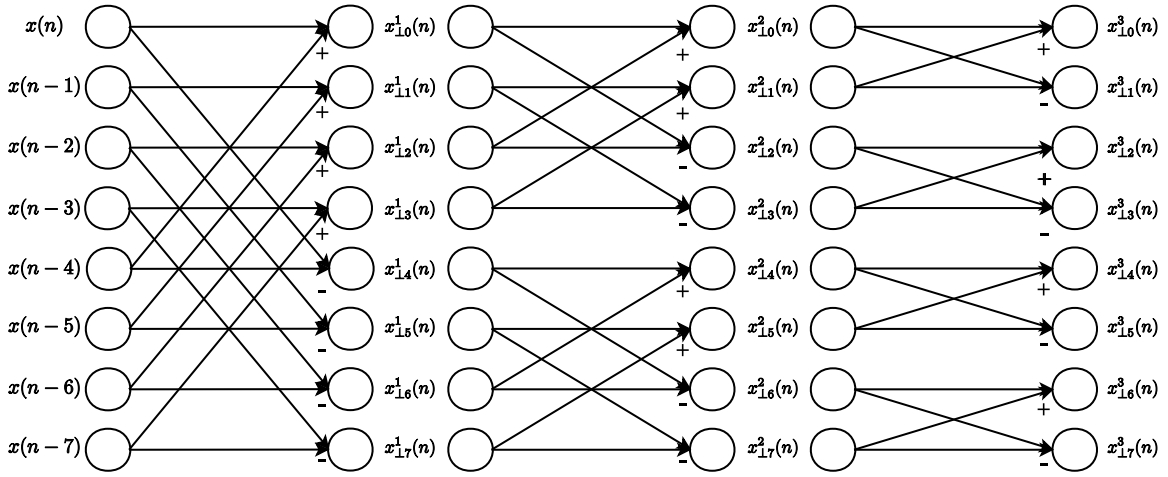

Figura 4.6: Esquema da Transformada Rápida de Hadamard

É possível ver que a saída do primeiro estágio divide o cálculo de uma WHT (*Walsh-Hadamard Transform* - Transformada de Walsh-Hadamard em português) de 8 pontos no cálculo de duas WHTs de 4 pontos. Este procedimento se repete na saída do segundo estágio, onde o cálculo de uma WHT de 4 pontos é subdividido de maneira que o resultado desta é calculado por duas WHTs de 2 pontos. Assim, a Figura 4.6 representa um fluxo completo para o cálculo de uma seqüência de dados com $N = 8$. A WHT de 2 pontos é o estágio básico de uma transformada rápida de Hadamard; também é conhecida como *butterfly*.

A transformada rápida de Hadamard é uma maneira computacionalmente mais eficiente de realizar os cálculos da transformada de Hadamard. Enquanto a transformada normal necessita de N^2 operações de adição e subtração para fornecer o resultado, a transformada rápida requer apenas LN adições e subtrações, $N = 2^L$. Como menos operações são necessárias, o espaço necessário para implementação da transformada de Hadamard em uma FPGA também é menor.

4.4 Efeito do procedimento multisplit na definição do hipercubo

A Tabela 4.1 apresenta os cálculos realizados pela Transformada Rápida de Hadamard para uma seqüência de dados.

Tabela 4.1: Cálculos realizados pela WHT

Seqüência de Dados	Estágio 1	Estágio 2	Estágio 3	Seqüência Transformada
0	0 + 4 = **4**	4 + 8 = **12**	12 + 16 = **28**	28
1	1 + 5 = **6**	6 + 10 = **16**	-16 + 12 = **-4**	-4
2	2 + 6 = **8**	-8 + 4 = **-4**	-4 + (-4) = **-8**	-8
3	3 + 7 = **10**	-10 + 6 = **-4**	4 + (-4) = **0**	0
4	-4 + 0 = **-4**	-4 + (-4) = **-8**	-8 + (-8) = **-16**	-16
5	-5 + 1 = **-4**	-4 + (-4) = **-8**	8 + (-8) = **0**	0
6	-6 + 2 = **-4**	4 + (-4) = **0**	0 + 0 = **0**	0
7	-7 + 3 = **-4**	4 + (-4) = **0**	0 + 0 = **0**	0

Relembrando, os coeficientes do filtro que utiliza o algoritmo LMS trigonométrico para a adaptação são relacionados a um conjunto de números A_k e a um ângulo θ_k, a partir do qual o valor $\sin(\theta_k)$ é calculado.

Como pode ser visto na Tabela 4.1, um dos efeitos do procedimento multisplit utilizando a transformada de Hadamard é o aumento da potência do sinal de entrada. Em várias aplicações, tais como [26] e [29], uma normalização da potência deste sinal é utilizada. Mas para o algoritmo LMS trigonométrico, é possível explorar esse aumento da potência para simplificar o procedimento de definição do hipercubo.

Como o ponto de ótimo é o produto da inversa da matriz de autocorrelação do sinal de entrada do equalizador pelo vetor de correlação cruzada entre o sinal desejado e o sinal de entrada, aumentando-se a variância do sinal de entrada o valor dos coeficientes que definem o ponto de ótimo diminui e, portanto, a definição do hipercubo

fica facilitada. Assim, pode-se considerar apenas o valor $\sin(\theta_k)$ para os coeficientes, fixando assim as coordenadas do hipercubo como unitárias e descartando a necessidade da definição de um hipercubo através de outros valores A_k.

Assim, as equações de filtragem e adaptação dos coeficientes no algoritmo LMS trigonométrico utilizando o procedimento multisplit podem ser reescritas como:

$$\theta(n+1) = \theta(n) + \mu\Delta(n)\mathbf{x}(n)e(n), \tag{4.60}$$

$$e(n) = d(n) - \sum_{k=0}^{N-1} \sin\theta_k(n)x(n-k), \tag{4.61}$$

onde Δ torna-se uma matriz diagonal com o j-ésimo elemento dado por:

$$\delta_{j,j} = \cos\theta_j, j = 0, 1, \ldots, N-1. \tag{4.62}$$

4.5 Conclusão

Neste capítulo foram apresentados o algoritmo LMS trigonométrico, o procedimento multisplit, a transformada de Hadamard que é utilizada por este procedimento e a utilização do procedimento multisplit para fixar as coordenadas do hipercubo que contém o ponto de mínimo da função objetivo. Como foi demonstrado através de simulações computacionais, este aumento na potência do sinal devido à utilização do procedimento multisplit facilita a definição das coordenadas do hipercubo. No próximo capítulo será apresentada a ferramenta utilizada para a implementação de projetos em FPGAs: o System Generator, da Xilinx.

Capítulo 5

System Generator

Nos capítulos anteriores, uma fundamentação teórica sobre o algoritmo CORDIC e suas várias aplicações foi apresentada. Mais detalhes sobre o equalizador adaptativo que utiliza o algoritmo LMS trigonométrico também foram fornecidos. Neste capítulo, será abordada a ferramenta que foi utilizada para a implementação deste sistema em uma FPGA. Esta ferramenta é o System Generator, da Xilinx, que trabalha em conjunto com o Matlab.

Nos últimos anos, as FPGAs se tornaram componentes muito importantes na implementação de sistemas de processamento digital de sinais (DSP) de alto desempenho, especialmente nas áreas de comunicações digitais, processamento de vídeo e imagem, e em redes de comunicações de dados. Entre as principais vantagens das FPGAs, podem-se citar a altíssima largura de banda da memória, a capacidade de implementação de arquiteturas completamente paralelas e a possibilidade de atualização dos equipamentos que os empregam [30].

Entretanto, a adoção das FPGAs historicamente tem sido lenta devido a vários motivos. Um deles é a falta de familiaridade com projeto de *hardware*. A maioria dos engenheiros de DSP trabalham com linguagens como o C ou o assembly, e desconhece linguagens de descrição de *hardware* (HDL) como o VHDL ou o Verilog. Além disso, linguagens como o VHDL fornecem muitas abstrações de alto nível para simulação, mas seu subconjunto sintetizável é muito restrito [30].

5.1 Fluxo de Projeto no System Generator

O System Generator é uma ferramenta de projeto para FPGAs da Xilinx, e funciona como uma biblioteca do Simulink (da Mathworks). Implementações em System Generator são fiéis, sintetizáveis e eficientes [30]. Uma das principais vantagens do System Generator é sua abstração aritmética, isto é, trabalha-se diretamente em ponto fixo com uma precisão arbitrária (incluindo os efeitos de quantização e *overflow* durante o projeto [31]). Com isso, evita-se a perda de precisão inerente à uma conversão de um número representado por precisão dupla para ponto fixo.

O Simulink fornece um ambiente gráfico para a criação e modelagem de sistemas dinâmicos. Já o System Generator consiste em bibliotecas do Simulink chamadas Xilinx Blockset e Xilinx Reference Blockset, e *software* para traduzir o modelo do Simulink em uma realização em *hardware* do modelo [30]. Os parâmetros definidos no Simulink para cada bloco são mapeados em portas, sinais, entidades, arquiteturas e atributos em uma realização de *hardware*. A Figura 5.1 representa o fluxo de desenvolvimento dentro do System Generator.

Os elementos do System Generator podem ser combinados livremente a outros blocos do Simulink. Entretanto, apenas os blocos do System Generator serão sintetizados para uma realização de hardware.

O System Generator pode ser útil em vários cenários. Pode ser utilizado para montar rapidamente um protótipo, a fim de possibilitar uma análise mais detalhada do modelo, estimar quantos recursos em *hardware* que ele irá ocupar ou seu desempenho. Também pode ser usado para implementar apenas parte de um projeto maior, deixando partes que necessitem de maior otimização para serem desenvolvidas utilizando HDL. E em muitos casos, o System Generator fornece todos os componentes necessários para o desenvolvimento de um projeto. Quando isso acontece, o System Generator traduz o projeto para HDL e gera os arquivos necessários para a interpretação do projeto.

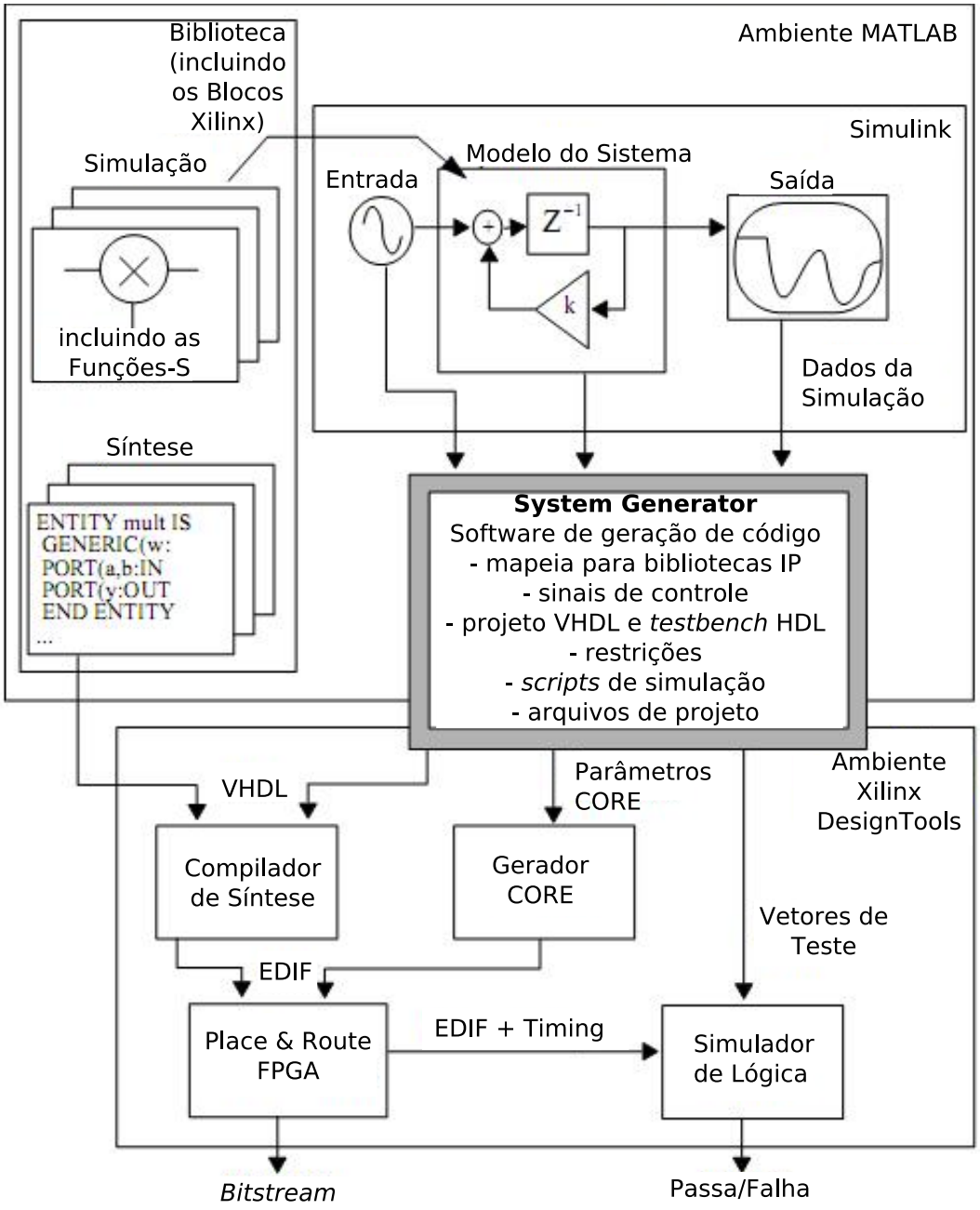

Figura 5.1: Fluxo de desenvolvimento do System Generator [30]

5.2 Blocos do System Generator

A biblioteca do System Generator é integrada à biblioteca do Simulink, e pode ser acessada por meio desta; é composta por vários blocos, que fornecem funções matemáticas, de controle, lógicas, de memória e de processamento digital de sinais, entre outras.

Além disso, é possível integrar código HDL ou Matlab a um sistema projetado com o System Generator, possibilitando assim a construção de sistemas sofisticados, sejam estes destinados a processamento digital de sinais ou a outras funções. Os blocos do System Generator são *bit-accurate* (produzem valores no Simulink que correspondem aos valores produzidos em *hardware*) e *cycle-accurate* (produzem valores correspondentes a tempos correspondentes). Por isso, é possível dizer que eles têm correspondência exata em *hardware*, incluindo suas características temporais [31].

As tabelas 5.1 e 5.2 enunciam as categorias principais de blocos do System Generator. A Figura 5.2 ilustra alguns dos blocos Xilinx do System Generator.

Tabela 5.1: Conjunto de Blocos Xilinx [31]

Biblioteca	**Descrição**
Índice	Cada bloco no conjunto de blocos Xilinx
Elementos Básicos	Padrão de blocos de formação para lógica digital
Comunicação	Blocos comumentes usados em sistemas de comunicação digital
Lógica de Controle	Blocos para controle de circuitos e estado de máquina
Tipos de Dados	Blocos de conversão de tipos de dados (incluindo portas)
DSP	Blocos de processamento digital de sinais
Matemática	Blocos que implementam funções matemáticas
Memória	Blocos que implementam e acessam memórias
Memória compartilhada	Blocos que implementam e acessam memórias compartilhadas Xilinx
Ferramentas	Geração de código (bloco *System Generator*, recursos de estimação, HDL(*Hardware Description Language*), co-simulação, etc

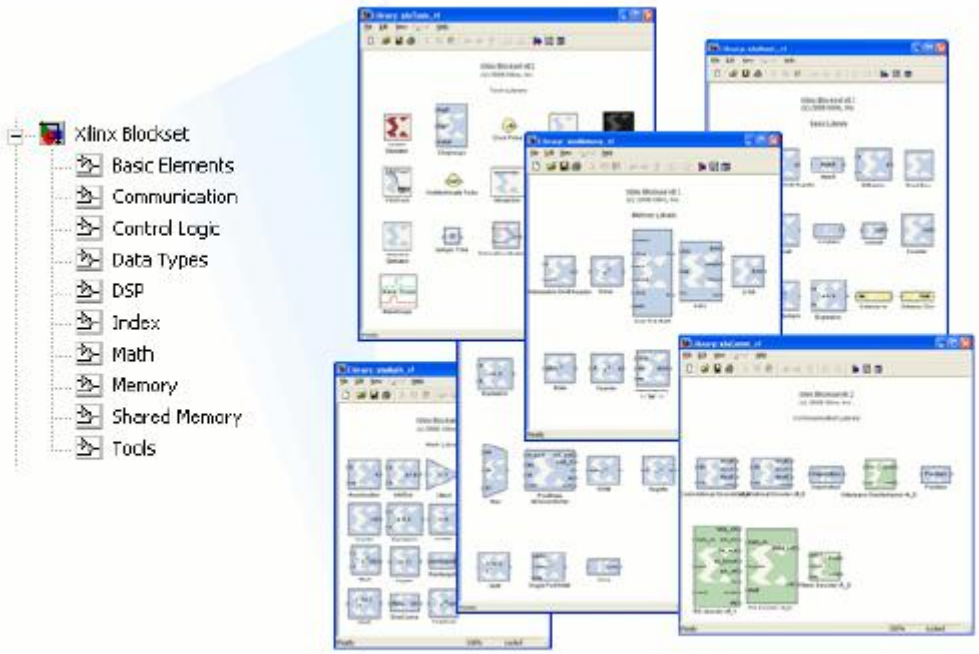

Figura 5.2: Conjunto de Blocos Xilinx

Tabela 5.2: Conjunto de Blocos de Referência Xilinx [31]

Comunicação	Blocos comumente usados em sistemas de comunicação digital
Lógica de Controle	Blocos para controle de circuitos e estado de máquina
DSP	Blocos de processamento digital de sinais
Imaging	Blocos de processamento de imagem
Matemática	Blocos que implementam funções matemáticas

5.3 Representação Aritmética

Para que o System Generator seja *bit-accurate*, seus blocos operam com valores booleanos e de ponto fixo (de precisão arbitrária). Já o Simulink opera com a representação dos sinais em ponto flutuante, com precisão dupla. A delimitação entre as áreas sintetizáveis e não-sintetizáveis é realizada pelos blocos do System Generator *Gateway in* (converte um sinal de ponto flutuante para ponto fixo) e *Gateway out* (realiza a operação inversa). A Figura 5.3 representa o esquema de delimitação entre as partes sintetizáveis e as não-sintetizáveis. A parte realçada corresponde aos blocos Xilinx,

que podem ser realizáveis em *hardware*.

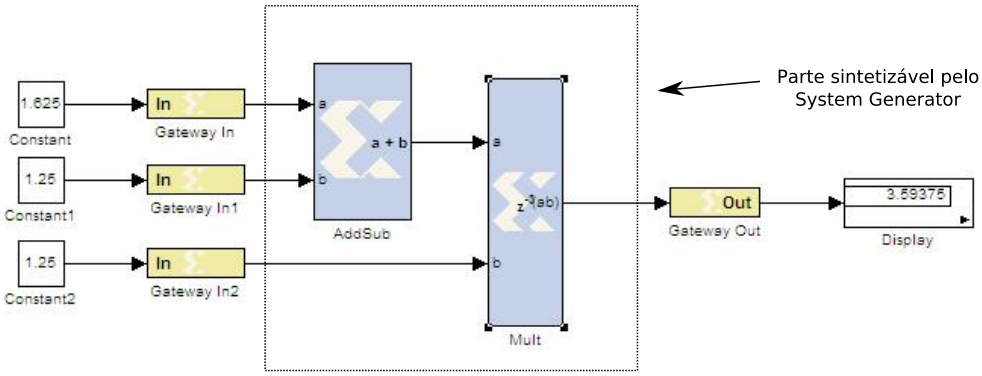

Figura 5.3: Conexão entre os blocos Xilinx e os blocos Simulink

Já os blocos individuais do System Generator podem ser configurados com precisão completa (o System Generator define o tipo de saída para garantir a precisão necessária) ou com a precisão definida pelo projetista, que também permite definir as configurações de *overflow*. Elas são: saturação (para o maior valor positivo ou menor valor negativo), truncamento (descartar os bits à direita do bit menos significativo) e geração de erro quando um *overflow* ocorre.

5.4 Temporização

Os sistemas projetados com o System Generator são de tempo discreto, isto é, cada bloco é associado a uma taxa de amostragem. O System Generator geralmente infere a taxa de amostragem para cada elemento automaticamente, mas caso seja necessário ou desejável é possível definir a taxa de amostragem individualmente. Também é possível projetar e sintetizar modelos que utilizem sinais com várias taxas de amostragem (multitaxa) [31]. O System Generator também inclui blocos que têm a função de mudar as taxas de amostragem. Estes blocos mudam a taxa de um sinal por um múltiplo inteiro que está especificado nas configurações do bloco. Outros blocos (ex.: os de conversão serial-paralela e paralela-serial) mudam as taxas implicitamente de uma maneira determinada pela parametrização do bloco. A Figura 5.4 mostra um modelo com duas taxas de amostragem, ilustradas pelas duas cores diferentes utilizadas nos contornos

dos blocos. O bloco *Down Sample* causa uma mudança de taxa, fazendo com que a parte delimitada em verde do modelo opere à metade da velocidade da parte vermelha.

Figura 5.4: Bloco de mudança de taxa [31]

Quando é dado o comando para a compilação de um modelo, o System Generator conserva as informações de taxa de amostragem do projeto, para que cada elemento funcione na taxa designada. Os períodos dos blocos individuais devem ser múltiplos inteiros do período do sistema. Esse período, juntamente com o período do relógio do FPGA, define o fator entre o tempo de simulação no Simulink e o tempo gasto na implementação real em *hardware*.

5.5 Geração Automática de Código

O System Generator produz automaticamente representações de baixo nível a partir do projeto desejado. As configurações do bloco *System Generator* controlam o resultado final da compilação e a geração ou não de arquivos auxiliares.

Entre as opções de compilação, é possível citar:

- dois tipos de *Netlists*: HDL e NGC;
- *bitstream*: produz uma *bitstream* pronta para rodar em uma FPGA;
- ferramenta de exportação para o EDK: para exportar para o EDK da Xilinx vários tipos de co-simulação por hardware;
- análise temporal: um relatório dos caminhos críticos do projeto, para que o projetista os possa otimizar e aumentar a velocidade de operação do sistema.
- co-simulação em *hardware*: o System Generator produz uma *bitstream* de configuração da FPGA que está pronta para rodar em uma plataforma FPGA. Ele

também produz um bloco de co-simulação de *hardware* a qual a *bitstream* está associada. Este bloco está habilitado a participar de simulações utilizando o Simulink. Nestas simulações, este bloco produz os mesmos resultados que o projeto a partir do qual o bloco de co-simulação foi gerado, mas os resultados são calculados utilizando a FPGA.

A Tabela 5.3 descreve os parâmetros de compilação que estão disponíveis quanto o tipo de compilação é HDL Netlist.

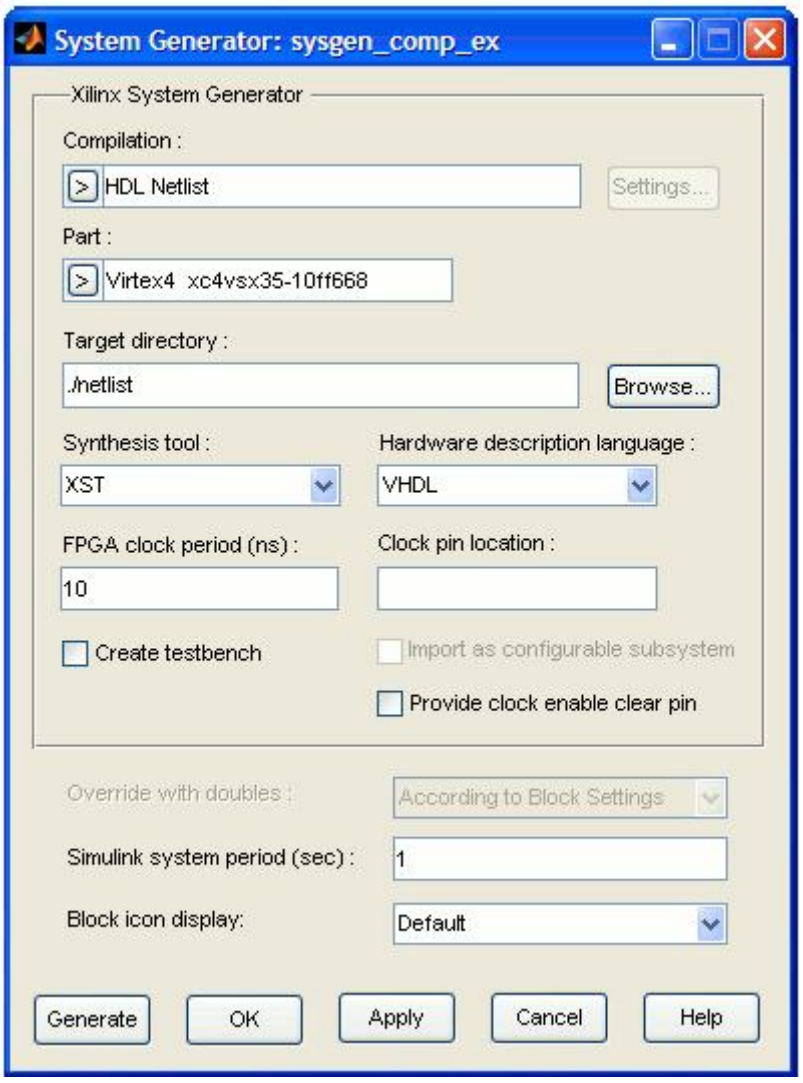

Figura 5.5: Opções do bloco System Generator

Tabela 5.3: Parâmetros de Compilação

Parte	Define o modelo de FPGA que será utilizado
Diretório Alvo	Define onde o System Generator gravará os resultados da compilação
Ferramenta de Síntese	Especifica onde a ferramenta que será usada na síntese do projeto; as possibilidades são: Synplicity, Synplify Pro, Synplicity e XST Xilinx
Linguagem de Descrição de Hardware	Especifica a linguagem que será usada pelo HDL Netlist no projeto; as possibilidades são: VHDL e Verilog
Período de *Clock* do FPGA	Define o período em nanossegundos do *clock* do hardware. O valor não precisa ser um número inteiro
Localização do pino de *clock*	Define a localização do pino para o *clock* do *hardware*
Cria *Testbench*	Essa instrução cria um HDL de teste
Importação como subsistema configurável	O System Generator realiza duas ações: construir um bloco o qual os resultados da compilação sejam associados; construir um subsistema configurável consistindo de um bloco e subsistema original, dos quais o bloco foi derivado.
Período de sistema Simulink	O valor indica a taxa em segundos que o sistema deve funcionar.

5.6 Estimação de Recursos

Com o bloco *Resource Estimator* do System Generator é possível estimar os recursos necessários da FPGA para o projeto em questão. Entre as estimativas, é possível encontrar o número de *slices*, LUTs, blocos de memórias, multiplicadores embarcados (já pré-definido em *hardware*), blocos de entrada/saída e *buffers* de três estados exigidos

pelo projeto. Tal recurso é muito importante, pois com ele é possível determinar se a FPGA utilizada suporta o projeto. A área ocupada pode ser estimada das seguintes maneiras:

- *Estimate*: carrega as funções de estimação de cada bloco e subsistema recursivamente;
- *Quick*: apenas lê a área especificada em cada bloco, nenhuma função de estimação é utilizada;
- *Post-map Area*: carrega a ferramenta de mapeamento de área Xilinx em todo o sistema (inclusive em blocos que não têm funções de estimação) e fornece os resultados do arquivo criado por esta ferramenta;
- *Read MRP*: para sistemas em que a ferramenta de mapeamento já foi utilizada.

A Figura 5.6 mostra um exemplo dos resultados que o *Resource Estimator* fornece.

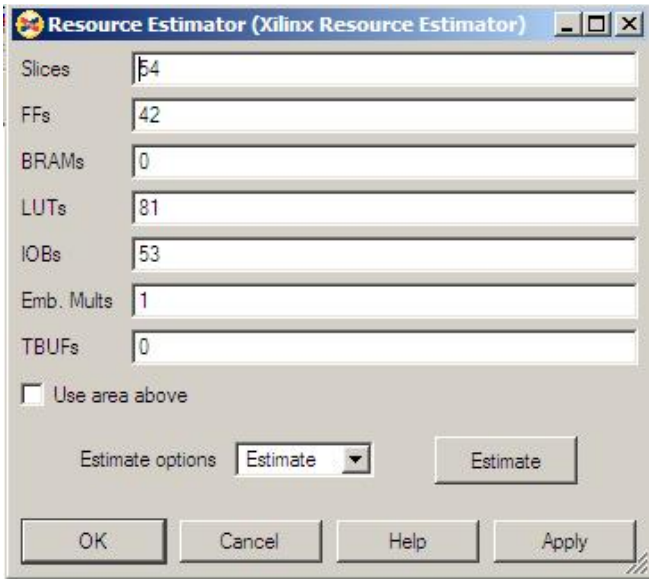

Figura 5.6: Resultados do *Resource Estimator*

5.7 Co-simulação em *hardware*

Utilizando o System Generator é possível gerar um bloco de co-simulação em *hardware* a partir de um projeto. Ele é equivalente ao projeto do qual foi gerado. Com este bloco, é possível utilizar os recursos da FPGA para as simulações realizadas no Simulink, com o mesmo resultado final das simulações sem a FPGA. O bloco criado para a co-simulação em *hardware* também pode ser salvo em uma biblioteca do Simulink para futuro uso. A Figura 5.7 ilustra o processo de co-simulação em *hardware*.

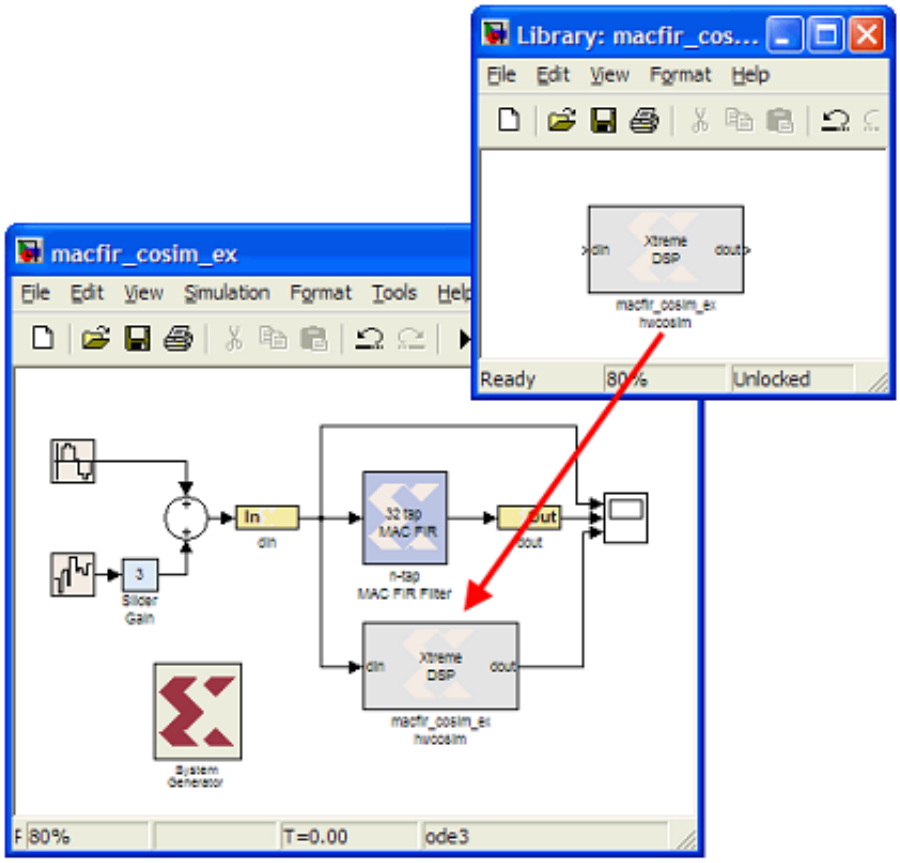

Figura 5.7: Bloco de co-simulação em *hardware* [31]

5.8 Conclusão

Neste capítulo foram apresentadas as principais características do System Generator, ferramenta que é instalada no conjunto de bibliotecas do Simulink e permite a re-

presentação fiel do *hardware* durante as várias fases do projeto. O próximo capítulo apresentará os equalizadores baseados em CORDIC que foram desenvolvidos com o auxílio do System Generator.

Capítulo 6

Implementação e Resultados

No capítulo anterior, a ferramenta a ser utilizada para a implementação dos equalizadores em FPGA, o System Generator, foi apresentada. Este capítulo inicia com detalhes sobre a arquitetura dos processadores CORDIC utilizados nos equalizadores adaptativos trigonométricos. Logo a seguir, a arquitetura dos equalizadores é detalhada, com informações sobre a área que estes ocupam em uma FPGA. Finalmente, os resultados das simulações são apresentados e comentados.

6.1 Processadores CORDIC

Existem dois principais tipos de processadores CORDIC: os iterativos e os paralelos. Os processadores CORDIC iterativos privilegiam a otimização da área ocupada, em detrimento da velocidade de operação. O contrário acontece com os processadores CORDIC paralelos.

6.1.1 Processadores Iterativos

Os processadores iterativos podem ser realizados de maneira simples: apenas três somadores/subtratores, três registradores de deslocamento e um dispositivo para armazenar a seqüência de deslocamento são necessários para realizar todas as operações de rotação requeridas pelo algoritmo. A seqüência de deslocamento pode ser armazenada em uma

ROM, RAM ou ser calculada no momento de cada iteração.

Na forma iterativa, o processador deve realizar as iterações a n vezes a velocidade desejada, diminuindo assim a taxa de amostragem com o crescimento do número de iterações desejadas do algoritmo. A Figura 6.1 mostra um processador CORDIC iterativo.

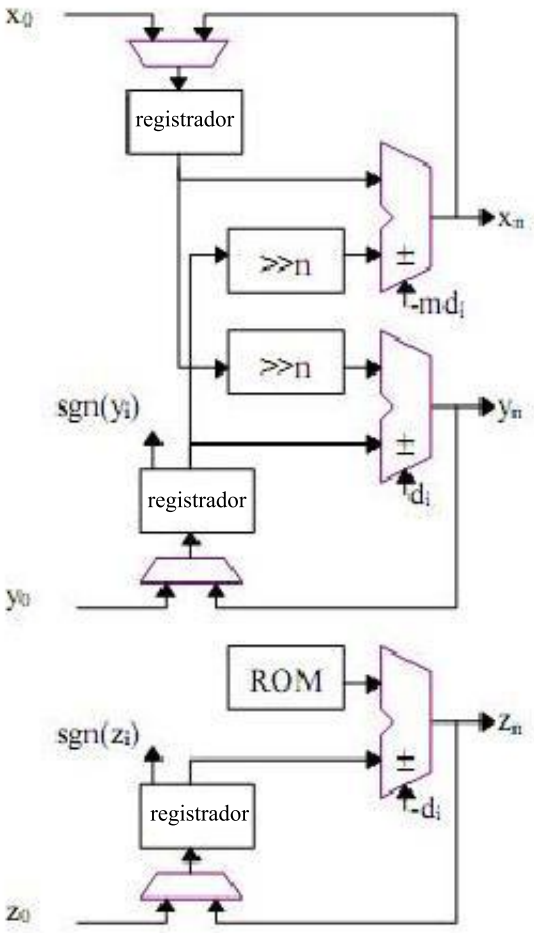

Figura 6.1: Processador Iterativo CORDIC [7]

6.1.2 Processadores Paralelos

Nos processadores paralelos, todas as operações de rotação do CORDIC podem ser realizadas em apenas um ciclo de relógio. Cada elemento de processamento é dedicado a uma iteração apenas. Cascateando estes elementos de processamento, é possível calcular todas as operações de rotação do CORDIC em apenas um ciclo de relógio [17]. Com

isso, duas simplificações importantes podem ser aplicadas. O deslocamento inerente às iterações do CORDIC fica fixo, podendo assim ser implementado diretamente nas conexões entre os elementos. E os valores do deslocamento do ângulo, que na forma iterativa são armazenados em algum dispositivo de memória, podem ser distribuídos como constantes para cada iteração.

Uma vantagem dos processadores paralelos é que entre cada iteração podem ser inseridos registros de *pipelining*, aumentando assim a velocidade de operação do processador [7]. Em muitas FPGAs, já há registros em cada célula lógica, fazendo com que o *pipelining* não consuma *hardware* adicional. A Figura 6.2 mostra um processador paralelo CORDIC *pipelined*:

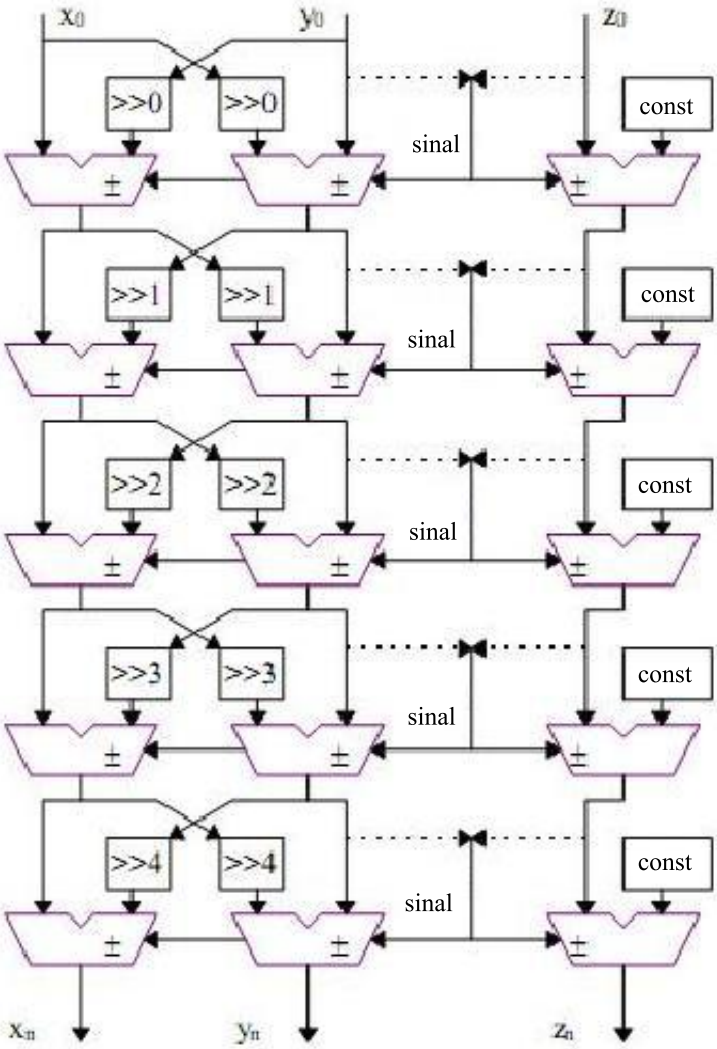

Figura 6.2: Processador Paralelo CORDIC *pipelined* [7]

6.2 Implementação

Pela flexibilidade oferecida por uma FPGA, a implementação de várias estruturas para realizar o processo de equalização é possível. Os fatores mais importantes para a escolha entre as arquiteturas são a taxa de amostragem requerida e o número de coeficientes necessário para o equalizador. Entre as estruturas disponíveis as mais comuns são a estrutura paralela e a estrutura seqüencial.

Na estrutura paralela, os coeficientes produzidos são somados ao mesmo tempo para gerar um resultado completo a cada ciclo de relógio. Em um equalizador totalmente paralelo, a máxima taxa de amostragem na entrada é igual à velocidade do relógio. A desvantagem da estrutura paralela comparada à estrutura serial é que ela requer um processador CORDIC para cada coeficiente do filtro.

Já a estrutura seqüêncial permite a utilização de apenas um processador CORDIC e um acumulador para todos os coeficientes, em vez de um para cada coeficiente. Entretanto, sua taxa de amostragem diminui à medida que o número de coeficientes do equalizador cresce. Por este motivo, a estrutura seqüêncial torna-se mais apropriada quando a taxa de amostragem necessária é baixa e o número de coeficientes deve ser grande.

A Figura 6.3 mostra como o aumento na taxa de amostragem e no número de coeficientes influencia na escolha de uma determinada estrutura.

Um modo de aumentar a taxa de amostragem da estrutura paralela do equalizador é fazer o *pipelining* da estrutura. Entretanto, este processo inevitavelmente diminui o desempenho do equalizador adaptativo, devido à maior latência na estrutura de realimentação. Essa latência faz com que um passo menor seja necessário para a estabilidade do algoritmo.

Além disso, a latência introduzida pelo *pipelining* faz com que o sinal de erro apenas esteja disponível algumas amostras depois. Por estes fatos, o algoritmo de adaptação utilizado no equalizador paralelo é o LMS trigonométrico atrasado.

A estrutura utilizada para gerar e atualizar cada coeficiente na estrutura paralela do equalizador utilizando o algoritmo LMS trigonométrico é apresentada na Figura 6.4.

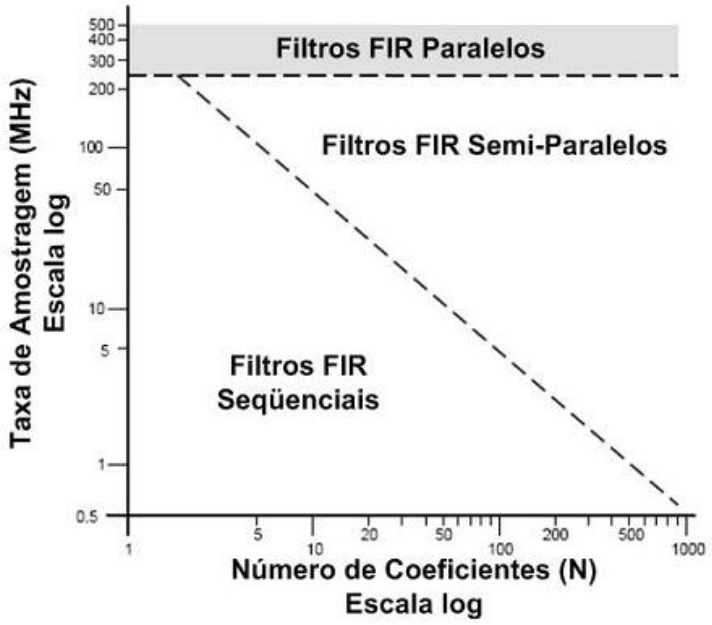

Figura 6.3: Comparação das estruturas dos filtros

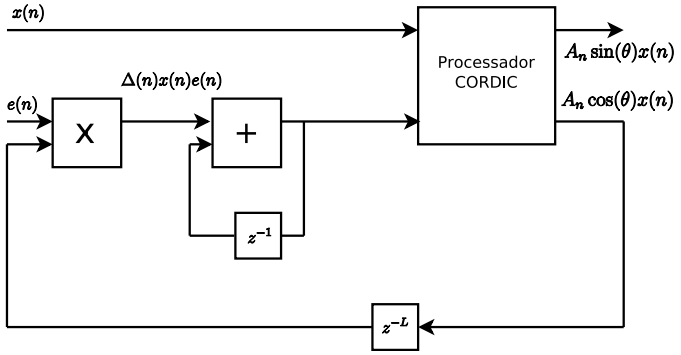

Figura 6.4: Estrutura para atualizar cada coeficiente na estrutura paralela

Como é possível ver, as operações de filtragem e adaptação dos coeficientes são realizadas por um processador CORDIC, e o cálculo da operação $\Delta(n)\mathbf{x}(n)e(n)$ é realizada por um multiplicador. O valor L corresponde à soma das latências do multiplicador e do processador CORDIC. A estrutura de somadores é da forma direta, como mostra a Figura 6.5.

Já a estrutura utilizada para a implementação seqüencial do equalizador utilizando o algoritmo LMS trigonométrico é apresentada na Figura 6.6. Esta arquitetura consiste em um processador CORDIC, uma memória RAM para armazenar os coeficientes e um multiplicador, que operam a uma velocidade N vezes maior do que a

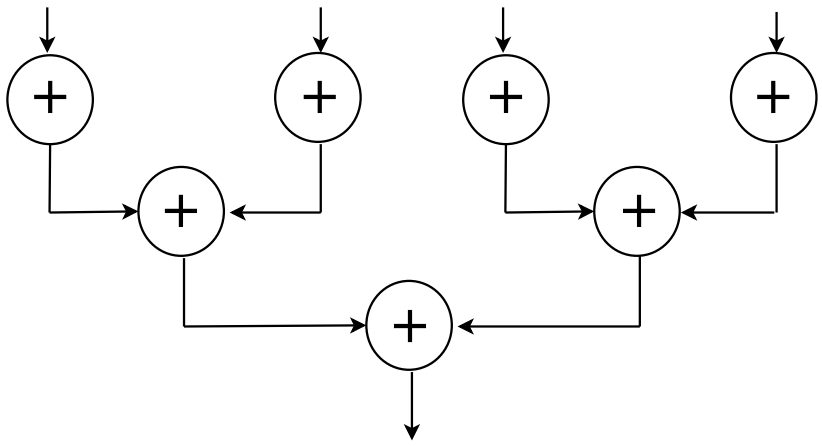

Figura 6.5: Estrutura de somadores em forma direta

entrada, onde N é o número de coeficientes. Tanto na entrada quanto na saída do processador CORDIC são colocados *Address Shift Registers* (ASR), que guardam as amostras de entrada e saída, enquanto o processador CORDIC está calculando a amostra presente. Entretanto, o ASR de entrada opera a uma taxa de f_s, que é a freqüência de amostragem, enquanto o ASR de saída opera a uma taxa de Nf_s.

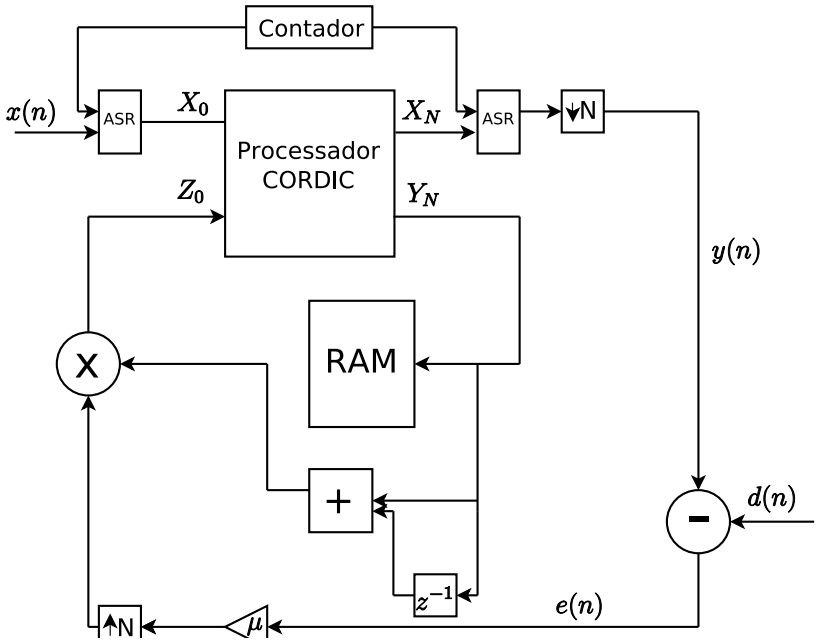

Figura 6.6: Estrutura utilizada para a implementação seqüencial do equalizador trigonométrico

Finalmente, o equalizador adaptativo que utiliza o algoritmo LMS trigonométrico com o procedimento multisplit é realizado de forma paralela, com a estrutura de filtra-

gem e atualização dos coeficientes igual à das figuras 6.4 e 6.5, mas em sua seqüência de entrada é realizada a transformada rápida de Hadamard, como foi indicado na Figura 4.6.

6.3 Resultados de Estimação de Recursos Ocupados na FPGA

A Tabela 6.1 mostra os resultados de estimação dos recursos ocupados para os equalizadores, cada um com 8 coeficientes, para uma FPGA Virtex 4 xc4vsx35-10ff668.

Tabela 6.1: Resultados de Estimação

Estrutura	Equalizador Seqüencial	Equalizador Paralelo	Equalizador Paralelo com Multisplit
Slices	1403	10670	12032
Flip Flops	588	11630	11686
BRAMs	1	0	0
LUTs	2610	19091	22427
IOBs	247	52	60
Bem. Mults.	0	0	0
TBUFs	0	0	0

Como é possível ver, o equalizador LMS trigonométrico seqüencial exige menos recursos e, conseqüentemente, ocupa uma menor área em FPGA que os outros equalizadores, mas sua taxa de amostragem máxima diminui à medida que o número de coeficientes aumenta, conforme a Figura 6.3. Também é possível inferir que o procedimento multisplit não aumenta de forma significativa os recursos necessários para o equalizador LMS trigonométrico paralelo.

6.4 Resultados

Para as seguintes simulações no System Generator, consideramos que um sinal 4-PAM está sendo transmitido por um canal real. As relações sinal-ruído utilizadas são 20 dB e 30 dB. Três algoritmos serão comparados: o LMS convencional, o LMS trigonométrico e o LMS trigonométrico utilizando o procedimento multisplit. Devido ao fato de o algoritmo LMS trigonométrico multisplit estar sendo implementado na estrutura paralela, o que força a utilização do LMS atrasado, os outros algoritmos também são utilizados em suas estruturas paralelas. Para o algoritmo LMS trigonométrico sem o procedimento multisplit, a busca de valores para o parâmetro A_k foi limitada a valores que são potência de 2. Assim, sua implementação em *hardware* pode ser realizada através de um simples deslocamento, que não consome recursos extras. Os processadores CORDIC utilizados foram implementados na forma paralela. Todos os equalizadores utilizados nas simulações têm oito coeficientes.

Os resultados da primeira simulação foram obtidos utilizando-se um canal de fase mínima com resposta à amostra unitária dada por: $\mathbf{h}_1 = [0,6080\ 0,0608\ -0,0486\ -0,1313]$. O diagrama de pólos e zeros deste canal é mostrado na Figura 6.7.

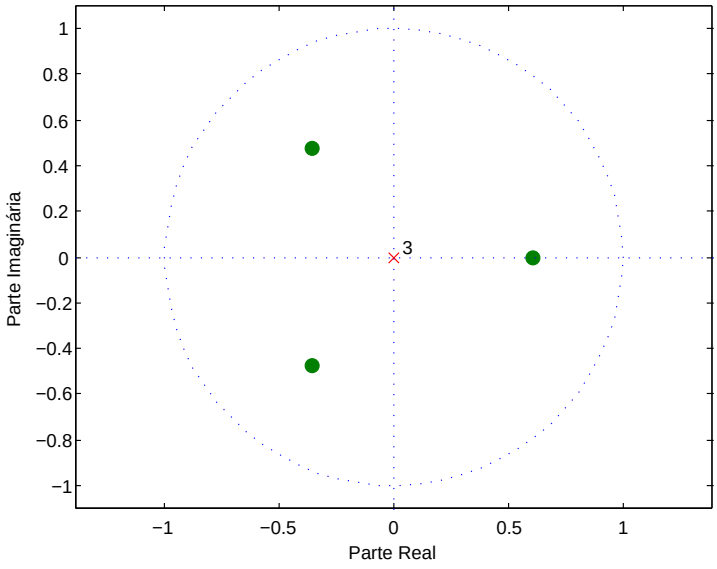

Figura 6.7: Pólos e zeros do canal utilizado pela Simulação 1

Foram mediadas 250 realizações para a construção das curvas de convergência

das Figuras 6.8 e 6.9. O passo de adaptação para o algoritmo LMS é $\mu_{LMS} = 0,009$. Já para o algoritmo LMS trigonométrico, o passo é $\mu_{TLMS} = 0,004$. Finalmente, para o algoritmo LMS trigonométrico utilizando o procedimento multisplit, o passo é $\mu_{TLMS-MS} = 0,0015$. O valor de A_k utilizado para o algoritmo LMS trigonométrico é 2.

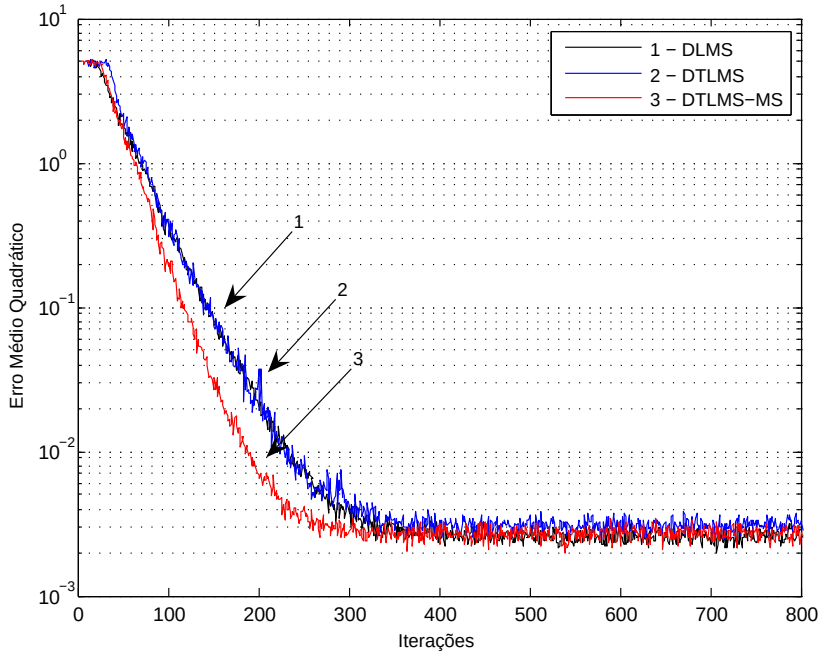

Figura 6.8: Curvas de aprendizado da Simulação 1 - 30 dB

É possível ver nas Figuras 6.8 e 6.9 que, para este canal, a velocidade de convergência do algoritmo LMS trigonométrico utilizando o procedimento multisplit é um pouco maior.

O canal de fase mínima $\mathbf{h}_2 = [0,8316\ 0,5325\ 0,4578]$ foi utilizado para a obtenção dos resultados da segunda rodada de simulações. Este canal tem seus zeros mais próximos ao círculo de raio unitário do que o canal utilizado na primeira simulação, como pode ser visto na Figura 6.10. O valor de A_k utilizado para o algoritmo LMS trigonométrico atrasado é 1. Os passos de adaptação utilizados para a Simulação 2 são: $\mu_{LMS} = 0,004$, $\mu_{TLMS} = 0,003$ e $\mu_{TLMS-MS} = 0,0005$. Novamente, 250 realizações foram realizadas para a construção das Figuras 6.11 e 6.12, que mostram as curvas de aprendizado da Simulação 2.

Como é possível ver nas Figuras 6.11 e 6.12, o algoritmo LMS trigonométrico

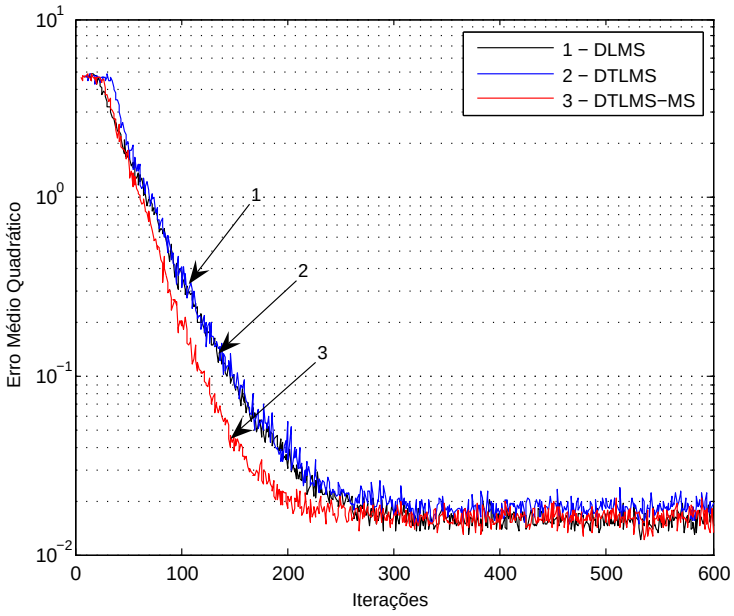

Figura 6.9: Curvas de aprendizado da Simulação 1 - 20 dB

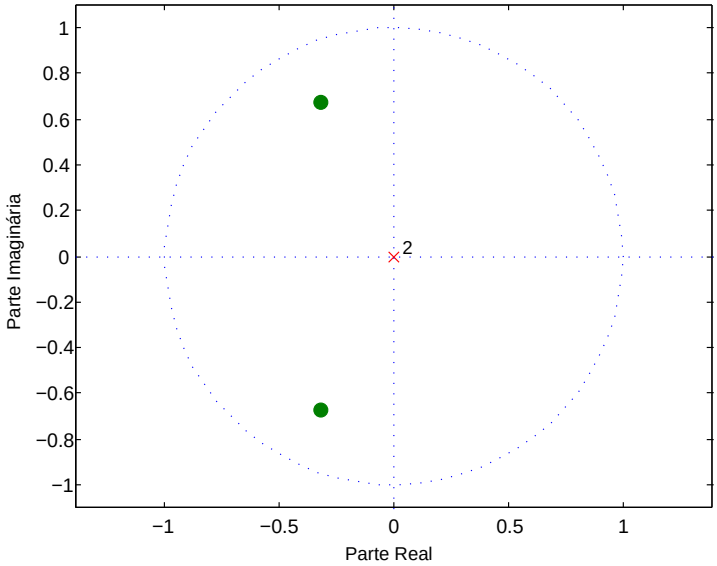

Figura 6.10: Pólos e zeros do canal utilizado pela Simulação 2

tem o erro médio quadrático na saída muito acima dos outros dois algoritmos, o que indica que o valor adotado para A_k está longe de um valor que permita uma boa convergência. Outros valores de A_k, como 0,5 e 2, fazem com que a convergência não seja alcançada. Para esta simulação, o desempenho dos outros algoritmos é similar.

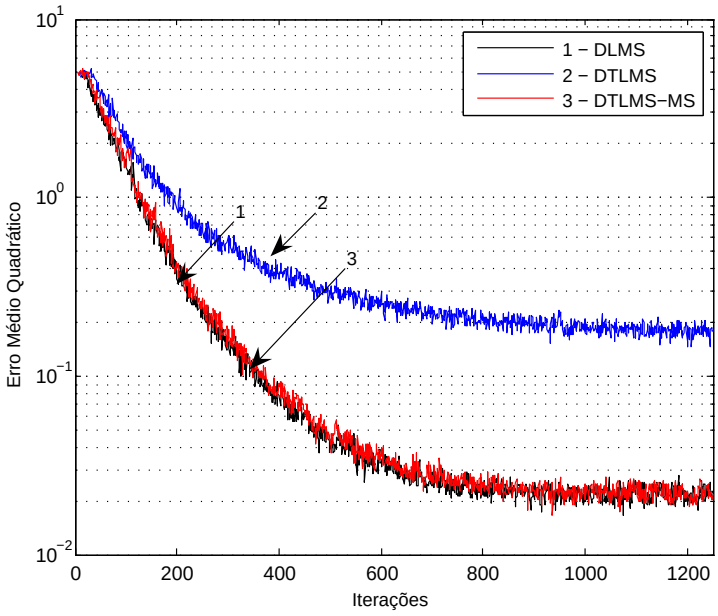

Figura 6.11: Curvas de aprendizado da Simulação 2 - 30 dB

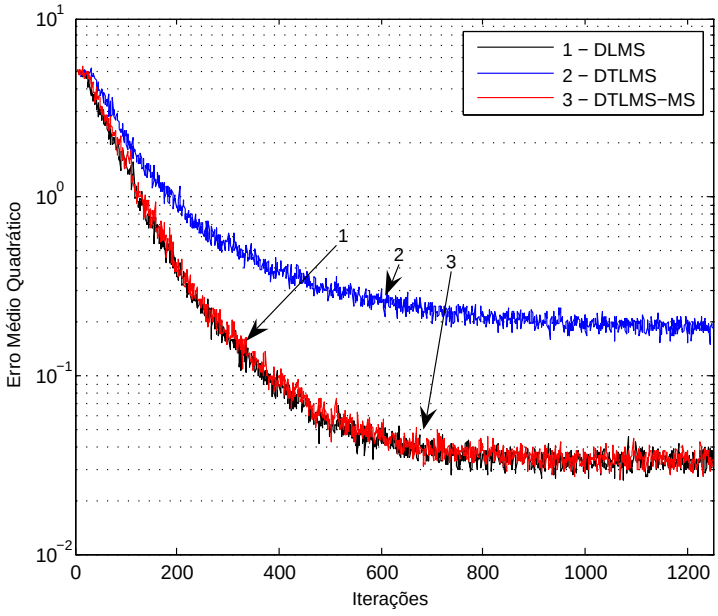

Figura 6.12: Curvas de aprendizado da Simulação 2 - 20 dB

Finalmente, apresentamos os resultados da terceira simulação, onde foi utilizado um canal de fase não-mínima com resposta à amostra unitária dada por $\mathbf{h}_3 = [-0,40\ 0,85\ 0,34\ 0,27]$. O padrão pólo-zero é apresentado na Figura 6.13.

Para esta simulação, os passos de adaptação são: $\mu_{LMS} = \mu_{TLMS} = 0.006$ e

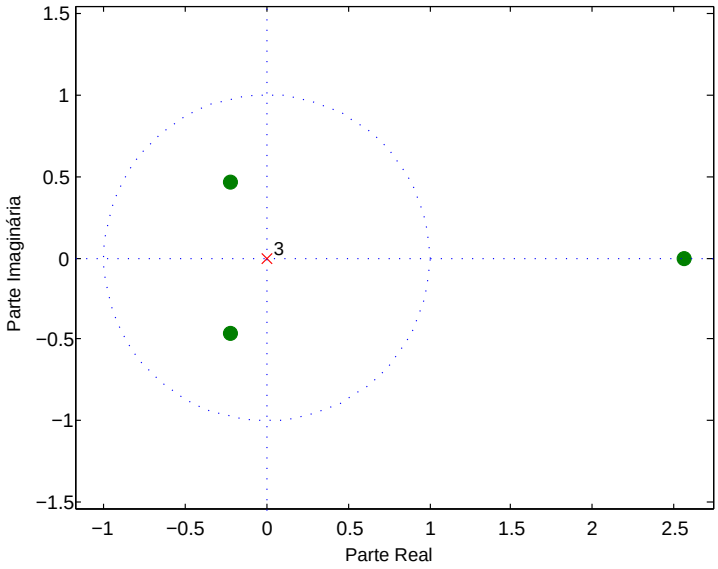

Figura 6.13: Pólos e zeros do canal utilizado pela Simulação 3

$\mu_{TLMS-MS} = 0.0008$. O valor de A_k para o algoritmo LMS trigonométrico é 1. As Figuras 6.14 e 6.15 apresentam as curvas de convergência para a Simulação 3.

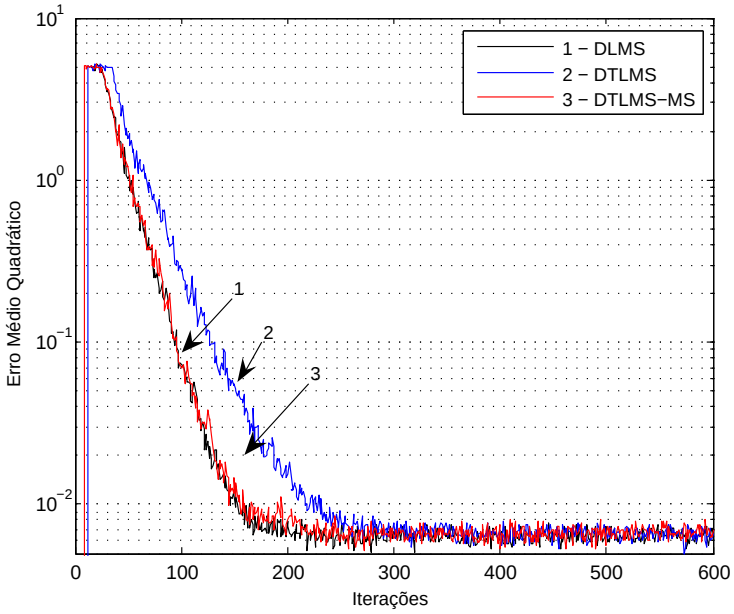

Figura 6.14: Curvas de aprendizado da Simulação 3 - 30 dB

Nessa simulação, o valor de A_k adotado permite uma boa convergência do algoritmo LMS trigonométrico, mas os algoritmos LMS e LMS trigonométrico com multis-

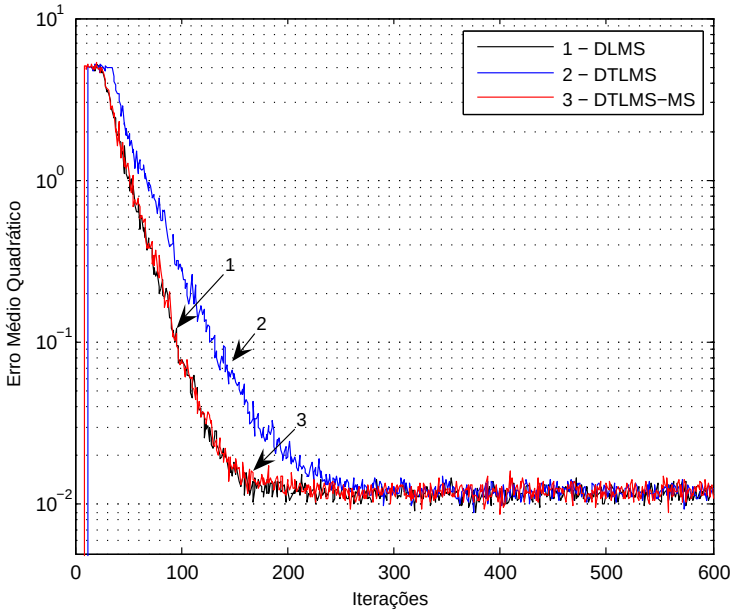

Figura 6.15: Curvas de aprendizado da Simulação 3 - 20 dB

plit convergem mais rapidamente.

É necessário ressaltar que o principal motivo para a adição do procedimento multisplit ao algoritmo LMS trigonométrico é a fixação das coordenadas do hipercubo, descartando-se assim a necessidade de uma busca exaustiva por um valor de A_k ótimo.

6.5 Conclusão

Nesse capítulo, foram apresentados os dois principais modos de implementação dos processadores CORDIC: os processadores iterativos e os processadores paralelos. Em seguida, as estruturas seqüencial, paralela e paralela utilizando o procedimento multisplit dos equalizadores adaptativos trigonométricos foram definidas. Este capítulo foi concluído com os resultados de estimação de recursos ocupados na FPGA e resultados de simulações em System Generator dos equalizadores. Essas simulações justificam a utilização do procedimento multisplit com o equalizador adaptativo trigonométrico.

Capítulo 7

Conclusões e Trabalhos Futuros

7.1 Conclusões Finais

O Capítulo 2 teve como foco o algoritmo CORDIC. Foram apresentados seu histórico, o desenvolvimento de suas três equações básicas, seus modos de operação e os métodos que podem ser utilizados para aumentar a faixa de operação do algoritmo. O objetivo principal desse capítulo foi prover ao leitor uma boa fundamentação teórica do algoritmo.

No Capítulo 3 foram apresentadas várias aplicações relacionadas a processamento de sinais e comunicações em que o algoritmo CORDIC pode ser empregado. Entre elas, mais detalhes foram fornecidos sobre a síntese digital direta (DDS), sincronização, modulação, *up/downconversion*, FFT e filtragem digital. Com isso, foi possível ilustrar a grande flexibilidade do algoritmo CORDIC.

Já o Capítulo 4 apresenta o foco da dissertação, que é a equalização adaptativa trigonométrica multisplit. O capítulo foi iniciado apresentando o algoritmo LMS trigonométrico. Depois é apresentado o procedimento multisplit, que utiliza a transformada de Hadamard. Esta transformada é mais bem detalhada na seção seguinte. Finalmente, a principal vantagem da aplicação do procedimento multisplit ao algoritmo LMS trigonométrico, que é a fixação das coordenadas do hipercubo que contém o ponto de mínimo da função objetivo, foi discutida.

No Capítulo 5, a ferramenta utilizada para a implementação em FPGA dos

equalizadores expostos no Capítulo 3, o System Generator, da Xilinx, foi abordada. Essa ferramenta trabalha em conjunto com o Matlab. Detalhes sobre o fluxo de projeto utilizando o System Generator, os blocos que são utilizados para a implementação de sistemas em FPGA e sua representação aritmética são fornecidos. O capítulo é finalizado com informações sobre a compilação dos sistemas projetados e sua co-simulação em *hardware*.

Finalmente, o Capítulo 6 apresenta uma breve discussão sobre os tipos possíveis de implementação de processadores CORDIC. Seguem mais detalhes sobre a estrutura dos equalizadores adaptativos implementados no System Generator e uma estimativa da área ocupada por eles. O Capítulo 6 termina com os resultados das simulações realizadas, que mostram que o algoritmo LMS trigonométrico utilizando o procedimento multisplit descarta a necessidade da busca de um valor de A_k que permita uma convergência ótima, convergindo assim para uma variedade de canais.

As principais contribuições deste trabalho são: a implementação em FPGA do algoritmo LMS trigonométrico e a utilização do procedimento multisplit que facilita a definição do hipercubo utilizado pelo algoritmo LMS trigonométrico.

Os resultados deste trabalho deverão ser submetidos para publicação no primeiro semestre deste ano.

7.2 Trabalhos Futuros

Como sugestões para a continuidade desse trabalho, podemos citar:

- implementação do algoritmo LMS trigonométrico utilizando o procedimento multisplit de forma seqüencial;
- expansão do algoritmo LMS trigonométrico para canais e constelações complexas;
- otimização do processador CORDIC, para uma maior velocidade de funcionamento e uma menor área ocupada na FPGA;
- implementação de uma forma otimizada em espaço da Transformada Rápida de Hadamard;

- expansão do algoritmo LMS trigonométrico para utilizar funções hiperbólicas;
- análise estatística do algoritmo LMS trigonométrico;
- inclusão de uma normalização de potência com baixo custo em *hardware* (por exemplo, utilizando apenas deslocamentos) no algoritmo LMS trigonométrico utilizando o procedimento multisplit.

Referências Bibliográficas

[1] J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, vol. EC-8, pp. 330–334, September 1959.

[2] Y. H. Hu and H. E. Liao, "CALF: A CORDIC Adaptive Lattice Filter," *IEEE Transactions on Signal Processing*, vol. 40, pp. 990–993, April 1992.

[3] S. ichi Shiraishi, M. Haseyama, and H. Kitajima, "An Implementation of a Normalized ARMA Lattice Filter with a CORDIC Algorithm," *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, vol. 5, pp. 253–256, May 1998.

[4] M. Chakraborty and A. S. D. anda Moon Ho Lee, "A Trigonometric Formulation of the LMS Algorithm for Realization on Pipelined CORDIC," *IEEE Transactions on Circuits and Systems - II: Express Briefs*, vol. 52, pp. 530–534, September 2005.

[5] J. E. Volder, "The Birth of CORDIC," *Journal of VLSI Signal Processing*, vol. 25, pp. 101–105, June 2000.

[6] J. S. Walther, "A Unified Algorithm for Elementary Functions," *Spring Joint Computer Conference*, pp. 379–385, April 1971.

[7] R. Andraka, "A survey of CORDIC algoritms for FPGA based computers," *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 191–200, February 22-24 1998.

[8] J. Valls, T. Sansaloni, A. Pérez-Pascual, V. Torres, and V. Almenar, "The Use of CORDIC in Software Defined Radios: A Tutorial," *IEEE Communications Magazine*, pp. 46–50, September 2006.

[9] J. Tierney, C. Rader, and B. Gold, "A digital frequency synthesizer," *IEEE Transactions on Audio and Electroacoustics*, vol. 19, pp. 48–57, March 1971.

[10] L. Cordesses, "Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1)," *IEEE Signal Processing Magazine*, pp. 50–54, July 2004.

[11] J. Vankka, "Methods of Mapping from Phase to Sine Amplitude in Direct Digital Synthesis," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 44, pp. 526–534, March 1997.

[12] C. Dick, F. Harris, and M. Rice, "FPGA Implementation of Carrier Synchronization for QAM Receivers," *The Journal of VLSI Signal Processing*, vol. 36, pp. 57–71, January 2004.

[13] E. B. Hogenauer, "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-29, pp. 155–162, April 1981.

[14] J. Vankka, M. Kosunen, I. Sanchis, and K. A. I. Halonen, "A Multicarrier QAM Modulator," *IEEE Transactions on Circuits and Systems - II: Analog and Digital Signal Processing*, vol. 47, pp. 1–10, January 2000.

[15] A. M. Despain, "Fourier Transform Computers Using CORDIC Iterations," *IEEE Transactions on Computers*, vol. C-23, pp. 993–1001, October 1974.

[16] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, pp. 297–301, April 1965.

[17] Y. H. Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing," *IEEE Signal Processing Magazine*, pp. 16–35, July 1992.

[18] R. Sarmiento, F. Tobajas, V. de Armas, R. Esper-Chain, J. F. Lopez, J. A. Montiel-Nelson, and A. Nuñez, "A CORDIC Processor for FFT Computation and Its Implementation Using Gallium Arsenide Technology," *IEEE Transactions on VLSI Systems*, vol. 6, pp. 18–30, March 1998.

[19] M. Garrido and J. Grajal, "Efficient Memoryless CORDIC for FFT Computation," *IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 2, pp. 113–116, April 2007.

[20] J. Burg, *Maximum Entropy Spectral Analysis*. PhD thesis, Stanford University, 1975.

[21] A. Goldsmith, *Wireless Communications*. Cambridge University Press, 2006.

[22] G. Long, F. Ling, and J. G. Proakis, "The LMS Algorithm with Delayed Coefficient Adaptation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37, pp. 1397–1405, September 1989.

[23] P. Kabal, "The Stability of Adaptive Minimum Mean Square Error Equalizeres Using Delayed Adjustment," *IEEE Transactions on Communications*, vol. 31, pp. 430–432, March 1983.

[24] L. K. Ting, R. Woods, and C. F. N. Cowan, "Virtex FPGA Implementation of a Pipelined Adaptive LMS Predictor for Electronic Support Measures Receivers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, pp. 86–95, January 2005.

[25] K. Wan and P. C. Ching, "Multilevel split-path adaptive filtering and its unification with discrete Walsh transform application," *IEEE Transactions on Circuits and Systems II*, vol. 44, pp. 147–151, February 1997.

[26] L. S. Resende, J. M. T. Romano, and M. Bellanger, "Split Wiener Filtering With Application in Adaptive Systems," *IEEE Transactions on Signal Processing*, vol. 52, pp. 636–644, March 2004.

[27] L. Griffiths and C. Jim, "An alternative approach to linearly constrained adaptive beamforming," *IEEE Transactions on Antennas and Propagation*, vol. AP-30, pp. 27–34, January 1982.

[28] L. S. Resende, *Algoritmos Recursivos de Mínimos Quadrados para Processamento Espacial/Temporal com Restrições Lineares: Aplicação em Antenas Adaptativas*. PhD thesis, UNICAMP, 1996.

[29] F. J. A. de Aquino, C. A. F. da Rocha, and L. S. Resende, "Equalização de Canal Usando um Algoritmo LMS Largamente Linear Multi Split," *XXV Simpósio Brasileiro de Telecomunicações - SBrT 2007*, September 2007.

[30] Xilinx, *Xilinx System Generator v2.1 for Simulink*.

[31] Xilinx, *System Generator for DSP: User Guide - Release 9.2.01*, October 2007.